OULUN YLIOPISTO
UNIVERSITY of OULU

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Sakari Alapuranen

Performance Optimizations for LTE User-Plane L2 Software

Master's Thesis
Degree Programme in Computer Science and Engineering
April 2015

Alapuranen S. (2015) Performance Optimizations for LTE User-Plane L2 Software. University of Oulu, Department of Computer Science and Engineering. Master's Thesis, 67 p.

ABSTRACT

Nowadays modern mobile communication networks are expected to be able to compete with wired connections in both latency and speed. This places a lot of pressure on the mobile communication protocols, which are very complex, and much of their implementation depends on the software. The performance of the software directly affects the capacity of the network, which in turn affects the throughput and latency of the network's users and the number of users the network can support.

This thesis concentrates on identifying software components of LTE User-Plane radio interface protocols for improvements, and exploring the solutions for better performance. This study leans on system component tests and the performance profiler tool *perf*, which enables tracking the effects of software optimizations from function-level to the whole system-level accuracy. In addition to *perf*, performance counters provided by the processor are manually observed and they provide the verification on why specific optimizations affect the performance.

Slow memory accesses or cache misses are identified as the most constraining factor in the software's performance. Also many good practices are found during the optimization work, such as arranging code common path first. Surprisingly, separating hardly executed code from hotspots also has a positive impact on performance, in addition to shrinking the active binary. The optimization work results in the whole software's load decreasing from 60% to 50% and in some individual functions load decreases of over 70% are achieved.

Keywords: mobile communication protocols, software optimizations, software performance, performance profiling, *perf*

Alapuranen S. (2015) LTE User-Plane L2-ohjelmiston suorituskykyoptimoinnit. Oulun yliopisto, tietotekniikan osasto. Diplomityö, 67 s.

TIIVISTELMÄ

Nykyään oletetaan, että modernit langattomat mobiiliverkot pystyvät kilpailemaan langallisten verkkojen kanssa sekä latenssissa että datansiirtonopeudessa. Tämä asettaa paljon haasteita langattomien mobiiliverkkojen protokollille, jotka ovat hyvin kompleksisia, ja paljon niiden implementaatiosta riippuu ohjelmistosta. Protokollien ohjelmiston suorituskyky vaikuttaa suoraan verkon kapasiteettiin, joka vuorostaan vaikuttaa käyttäjien datansiirtonopeuksiin ja latenssiin sekä siihen, kuinka montaa käyttäjää verkko pystyy tukemaan.

Tämä diplomityö keskittyy tutkimaan sekä toteuttamaan suorituskykyä parantavia ratkaisuja LTE User-Plane L2-protokollien ohjelmistoon. Työssä käytetään systeemikomponenttitason testejä sekä suorituskyvyn profilointityökalua *perf* varmentamaan ohjelmistoon tehdyt optimoinnit. *Perf* pystyy profiloimaan ohjelmiston sekä funktio- että systeemitasolla. *Perf*:n lisäksi prosessorin tarjoamia suorituskykylaskureita seurataan manuaalisesti ja ne tarjoavat selityksen miksi tietyt optimoinnit vaikuttavat ohjelmiston suorituskykyyn.

Välimuistin ohittavat muistihaut tunnistetaan olevan ohjelmiston suorituskyvyn rajoittavin tekijä. Monia hyviä käytäntöjä löydetään optimointityön aikana kuten koodin järjestely yleisin polku ensin. Harvoin suoritetun koodin erottaminen ohjelmiston jatkuvasti suoritetuista kohdista huomataan myös tuottavan yllättävän positiivinen vaikutus ohjelmiston suorituskykyyn. Optimointityön alkaessa ohjelmiston tuottamaa kuorma oli 60 % ja lopussa 50 %. Muutamat yksittäiset funktiot pystyttiin optimoimaan niin, että niiden kuorma laski alkuperäisestä kuormasta jopa 70 %.

Avainsanat: mobiiliverkkojen protokollat, ohjelmiston optimoinnit, ohjelmiston suorituskyky, suorituskyvyn profilointi, *perf*

TABLE OF CONTENTS

FOREWORD

I would like to thank my supervisor Prof. Olli Silvén for being extremely motivated and helpful during the whole process of the thesis work. I'm also extremely grateful for the steering group members Ms. Virpi Hanni, Mr. Juha Toivonen, Mr. Juha Kangas and Mr. Jouko Lindfors from Nokia Networks for giving me the opportunity to write this thesis and for their support and feedback in all matters related to it.

Big thanks to my team mates and co-workers in LTE UP Feature Team 9. Your feedback and guidance in the optimizations work was essential. Special thanks to Mr. Petri Laine and Mr. Juha Toivonen for their help and extensive knowledge in everything related to optimizations and LTE User-Plane L2 software.

Finally I would like to thank my parents and girlfriend Niina for believing in me during the thesis work and in my studies overall. Your support carried me through times when I didn't have much faith in my own success. Thank you.

Oulu, 15.4.2015

Sakari Alapuranen

ABBREVIATIONS

GSM	Global System for Mobile Communications
2G	Second generation mobile networks
3G	Third generation mobile networks
HSDPA	High Speed Downlink Access
4G	Fourth generation mobile networks
LTE	Long Term Evolution
WiMAX	Worldwide Interoperability for Microwave Access
LTE-Advanced	Long Term Evolution Advanced
DL	Downlink
UL	Uplink
eNodeB, eNB	Evolved Node B
UE	User Equipment
L1	Layer-1
L2	Layer-2
MAC	Medium Access Layer
RLC	Radio Link Control
PDCP	Packet Data Convergence Protocol
E-UTRAN	Evolved Universal Terrestrial Radio Access Network
EPC	Evolved Packet Core
MM	Mobility Management
EPS	Evolved Packet System
SRB	Signaling Radio Bearer
DRB	Data Radio Bearer
ROHC	Robust Header Compression
PDU	Protocol data unit
SDU	Service data unit
ARQ	Automatic repeat request
TM	Transparent Mode
UM	Unacknowledged Mode
AM	Acknowledged Mode
HARQ	Hybrid automatic repeat request
UMD	Unacknowledged Mode data
TB	Transport Block
CPU	Central processing unit
RTCG	Runtime code generation
VSO	Value-specific optimization
IPO	Interprocedural optimizations
OoOE	Out-of-order execution
ILP	Instruction level parallelism
CMP	Cache Chip Multiprocessor
LRU	Least-recently used
EM	Event Machine
EO	Execution Object
SCT	System component tests
PMU	Performance monitoring unit
SCM	Software configuration management

1. INTRODUCTION

Previously in mobile network systems, performance optimizations have focused on making improvements on hardware rather than software. This has caused performance optimizations to be relatively slow. Now these systems have been developed to be more software dependent, which has created the possibilities for performance optimizations in software. Software performance optimizations are usually less expensive and easier to implement compared to hardware performance optimizations. In this thesis different possibilities for performance optimizations are introduced and analyzed for LTE User-Plane L2 software.

1.1. Evolution of data traffic in mobile communications

The capability to carry data traffic over mobile networks was first introduced by the second generation mobile networks (2G) – such as Global System for Mobile Communications (GSM), General packet radio service (GPRS) and Enhanced Data rates for GSM Evolution (EDGE). In spite of this, 2G was dominated by voice traffic. The data capabilities of 2G networks depended on the standards, network architecture and transmission techniques, thus the goal of the software of 2G devices was to achieve the capabilities set by the standards and specifications. [1] [2]

With the third generation mobile networks (3G) High Speed Downlink Access (HSDPA) was introduced. It had the capability to handle much greater amounts of data traffic than the 2G networks and was one of the driving forces that changed mobile networks from voice dominated to packet data dominated networks. 3G introduced faster transmission techniques and improved network architecture compared to 2G, amongst other things. Even so, the software of 3G devices was limited by the standards and specification set for it. [1] [2]

Although the 3G networks were able to transfer data from and to users much faster than 2G networks, they were far behind the data traffic capabilities of the wireline networks. The fourth generation mobile networks (4G) were specified and developed to further increase the data traffic capabilities, therefore catching up to the wireline networks in that regard [1].

4G introduced first Long Term Evolution (LTE), WiMAX (Worldwide Interoperability for Microwave Access) and later Long Term Evolution Advanced (LTE-Advanced). LTE was more widely adopted than WiMAX as it offered higher bit rate, lower latency and many other service offerings compared to WiMAX and its predecessors [3]. LTE-Advanced is an evolution of LTE and it offers even higher bit rates, lower latencies and more features than LTE.

4G networks have fewer nodes in their architecture and less complexity in radio protocols compared to 3G networks. This provides the possibility to increase the performance of the network by improving software even after the specifications are met rather than upgrade hardware or revise the network architecture or techniques used in it, which has been the way this has been done before. Table 1 shows the evolution of mobile network technologies.

Table 1. Evolution of mobile network technologies

	LTE-Advanced(4G)	LTE(4G)	HSDPA(3G)	EDGE(2G)	GPRS(2G)
Modulation	QPSK, 16QAM, 64QAM	QPSK, 16QAM, 64QAM	QPSK, 16QAM, 64QAM	GMSK, 8PSK	GMSK
Multiple access schemes	OFDMA (DL), SC-FDMA (UL)	OFDMA (DL), SC-FDMA (UL)	CDMA	TDMA	TDMA
New features	8x8 MIMO, Carrier Aggregation	Flat architecture, 2x2 MIMO, VoLTE, Scalable bandwith, Compatible with 2G and 3G networks	Fast packet scheduling, HARQ, Adaptive modulation and coding, Dual-Cell	8PSK encoding	GMSK encoding

The theoretical peak for the downlink (DL) data rate with LTE-Advanced is 1 Gbps and for uplink (UL) the peak data rate is 500 Mbps [4]. This is much higher than any mobile network has been capable of before. The theoretical peak data rates are shown in Figure 1.

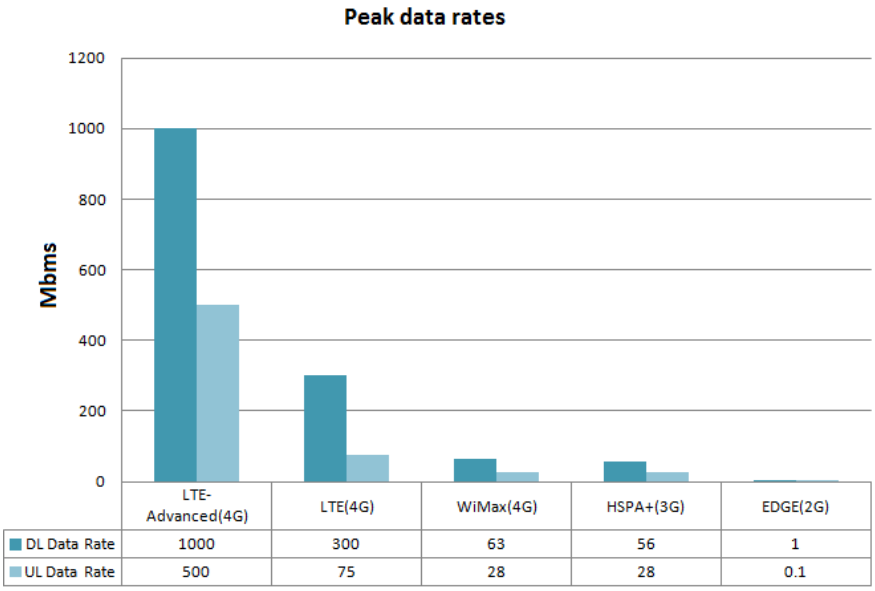

Figure 1. The theoretical peak data rates of different mobile networks.

LTE and later LTE-Advanced have also been able to reduce the network latency significantly, which is seen in Figure 2. It is obvious that the mobile networks have come a long way regarding data traffic, but the customer need for faster mobile networks is growing constantly.

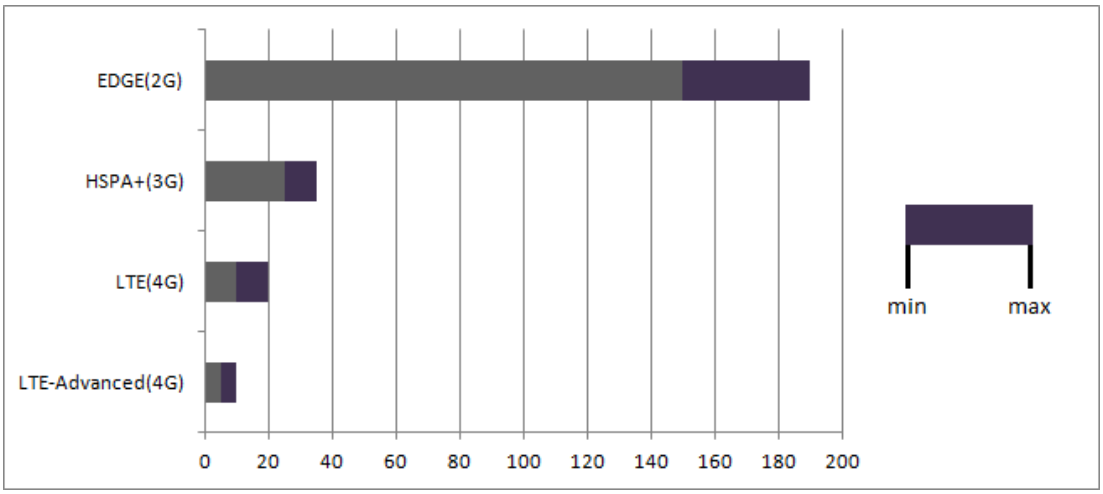

Figure 2. The theoretical minimum and maximum latencies of different mobile network technologies. (X-axis is time in milliseconds)

In 2013 the amount of global mobile devices grew to 7 billion [5]. It is predicted that the amount of mobile devices in use will surpass the 7.1 billion people on earth in the year 2014 [6]. Also according to Cisco, the global mobile data traffic will increase nearly 11-fold between 2013 and 2018 [5]. This creates great demand for capable mobile communication networks and the performance of those networks is of high importance.

1.2. Scope of the thesis

The Evolved Node B (eNodeB) is a LTE network element that communicates directly with user equipments (UEs). The LTE User-Plane radio interface protocols include the Layer-1 (L1) Physical Layer protocol as well as the Layer-2 (L2) protocols, i.e. Medium Access Layer (MAC), Radio Link Control (RLC) and Packet Data Convergence (PDCP) protocols.

In this thesis, we are interested in the performance of the LTE user plane protocol stack between the UE and eNodeB. These protocols and their functions are extensively described in Chapter 2.

All the data packets are processed by the PDCP, RLC and MAC before being passed to the physical layer for transmission [1]. This means that the functionalities in these protocols need to be executed efficiently and fast. The time spent in these protocols must be relatively low compared to the time it takes for the actual transmission of information. These protocols are complex and most of their implementation depends on the software. Also the 4G standard development & customer specific requirements create more pressure on software for continuous performance optimization. In this thesis the focus is on the PDCP, RLC and MAC layers of one eNodeB product and their performance characteristics, measurement and optimization.

1.3. Thesis structure

In Chapter 2 the Evolved Node B architecture and the role of the hardware and software in it is introduced. Chapter 3 goes into detail about different performance optimization methods. In Chapter 4 the performance evaluation framework and tools are described. Chapter 5 shows the results of the optimization work and goes into detail about a couple of successful optimization examples. Chapter 6 reflects upon the results of Chapter 5 and discusses optimization methods left unused in the thesis. Chapter 7 summarizes the thesis.

2. LTE ARCHITECTURE

LTE Architecture can be split into four main high level domains: User Equipment (UE), Evolved Universal Terrestrial Radio Access Network (E-UTRAN), Evolved Packet Core (EPC) and the Services domain [1]. Figure 3 provides a high-level view of LTE architecture.

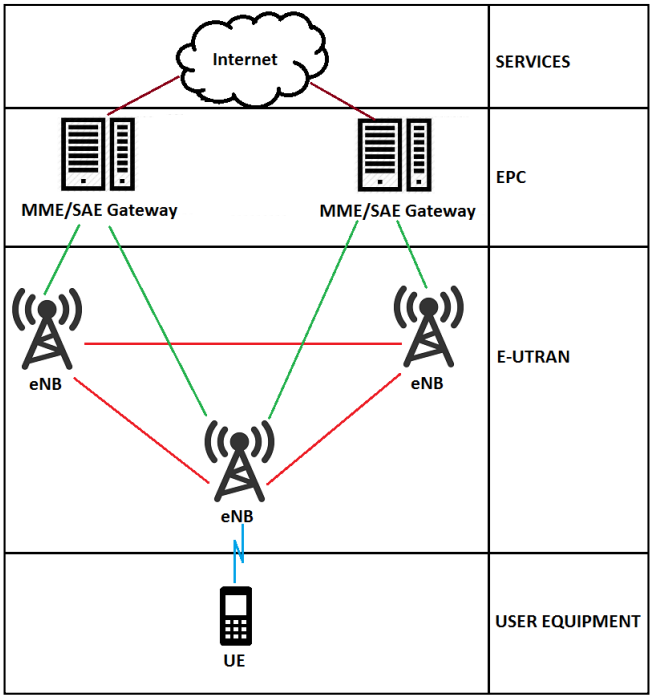

Figure 3. LTE Architecture overview.

The Services domain includes Operator Services and the Internet. The EPC is the core network of the LTE system. It connects the rest of the network to the Services domain. The EPC is an evolution of the packet-switched architecture used in 3G [7]. The E-UTRAN handles the communication between the UEs and the EPC and has one component, the eNodeB or eNB, which is another name for the base station [8]. The User Equipment domain contains all devices with the ability to connect to the LTE network. For example smart phones and tablet computers can be considered as user equipment.

2.1. LTE Advantages

The key difference between LTE and HSPA (3G) architecture is that there is no Radio Network Controller (RNC) in the LTE architecture. The RNC is responsible for controlling the Node Bs that are connected to it in 3G networks. In LTE the functionality of the RNC is included in the eNBs, which means that the LTE architecture has one node less than the HSPA architecture. This makes LTE architecture flat with only 2 nodes and naturally results in much faster communication between UEs and the network, which means lower latency and higher throughput.

The air interface in LTE is also different from previous generations as seen in Table 1. The multiple access schemes used in LTE are orthogonal frequency-division multiple access (OFDMA) and single-carrier frequency-division multiple access (SC-FDMA). OFDMA is used in downlink and SC-FDMA is used in uplink. OFDMA significantly increases spectral efficiency in LTE compared to HSPA and thus uses the bandwidth more efficiently and has more processing power. The reason why SC-FDMA is used in uplink is the high power consumption of OFDMA, which makes it unpractical to be used in the UEs.

The GSM architecture relies on circuit-switching (CS). This means that circuits are established between the calling and called parties throughout the telecommunication network. In GPRS and UMTS (3G) the architecture supports both packet-switching (PS) and CS. With PS technology data is transported in packets without the establishment of dedicated circuits.

The LTE architecture's interfaces are Internet Protocol (IP) based. IP is used for control and user plane as the network layer in the protocol stack of all interfaces. The EPC is an evolution of the PS architecture used in GPRS/UMTS and CS is no longer used. This offers more flexibility and efficiency. [7]

Figure 4 shows a simplified view of the LTE network. The figure highlights the importance of eNB in the LTE network. The whole LTE network depends on the capabilities of the eNB. Both the quality of software and hardware in eNB are of great importance. The faster the eNB operates, the lower the latency and the higher the throughput the user has. This chapter concentrates on the eNB and discusses its structures and complexities.

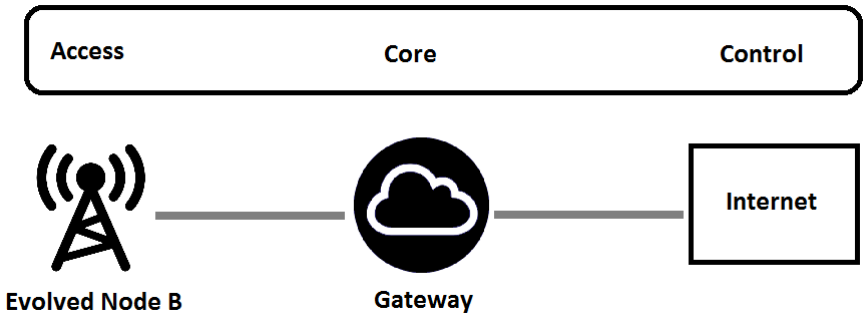

Figure 4. LTE: Flat, IP-based network.

2.2. The Evolved Node B

The eNodeB controls all radio related functions in the fixed part of the LTE system. Typically eNodeBs are distributed throughout the networks coverage area, each residing near the actual radio antennas. In short the eNodeB functions as a layer 2 bridge between the UE and the EPC. It is the termination point of all the radio protocols towards the UE, and it takes care of relaying data between the radio connection and the corresponding IP based connectivity towards the EPC. [1]

LTE, with its IP based and flat architecture, places much pressure on the eNodeB. At the functional level, the eNodeB needs to follow the 3GPP standard and be compliant with it. At the performance level, the eNodeB needs to support thousands of UEs, each with varying data rates. The high number of UEs per eNodeB, the high

data throughput and the low latency require the software and hardware of eNodeB to be optimal and well performing.

The eNodeB also has an important role in Mobility Management (MM), which means that the eNodeBs themselves are a part of the decision when to move a user under a different eNodeB. This transition usually happens when the user moves too far away from the eNodeB it is currently under. This chain of events is called a handover and is needed because a user can only be under one eNodeB at a time. [1]

The eNodeB is responsible for many Control-Plane functions, such as Mobility Management. The Control-Plane carries the control information of the network and the User-Plane carries the network's user's traffic.

The low latency of User-Plane is relevant nowadays for many applications such as real time gaming. Latency is measured by the time it takes for a small IP packet to travel from the UE to the internet server, and back. This is called round trip time measurement and is illustrated in Figure 5. Holma & Toskala assume that the eNodeB processing delay is 4ms and the whole round trip time of LTE network approximately 20ms. [1]

The 4ms delay target creates a high restriction for eNodeB's performance but also leaves room for improvement. If the eNodeB's functionalities could be performed faster, the round trip time for the whole LTE network would be less and the UE would experience less delay.

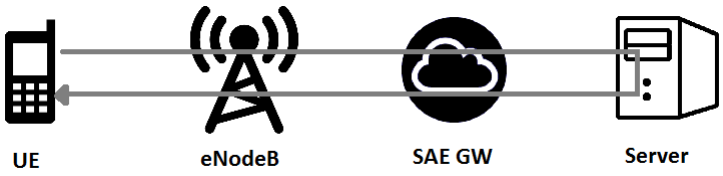

Figure 5. LTE round trip time measurement.

2.3. LTE L2 Radio Protocols

The role of the LTE radio interface protocols is to set up, reconfigure and release the Radio Bearer that provides the means for transferring the Evolved Packet System (EPS) bearer [1]. The LTE radio protocol stack is shown in Figure 6.

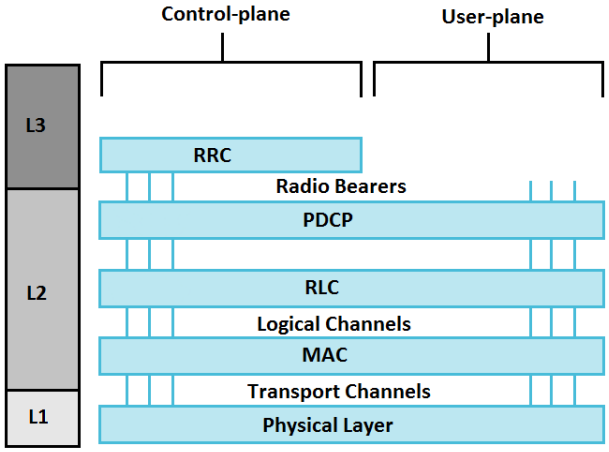

Figure 6. LTE Radio Protocol Stack Layers.

There are two types of Radio Bearers in LTE: Signaling Radio Bearer (SRB) and Data Radio Bearer (DRB). SRBs carry signaling messages and DRBs carry the user data. The term bearer can be defined in the communication system as a pipe line connecting two or more points. Therefore the EPS Bearer can be defined as a pipe line through which data traffic flows within an Evolved Packet Switched System that is the LTE network. The LTE User-Plane radio protocols between the eNodeB and the UE can be seen in Figure 7. [1]

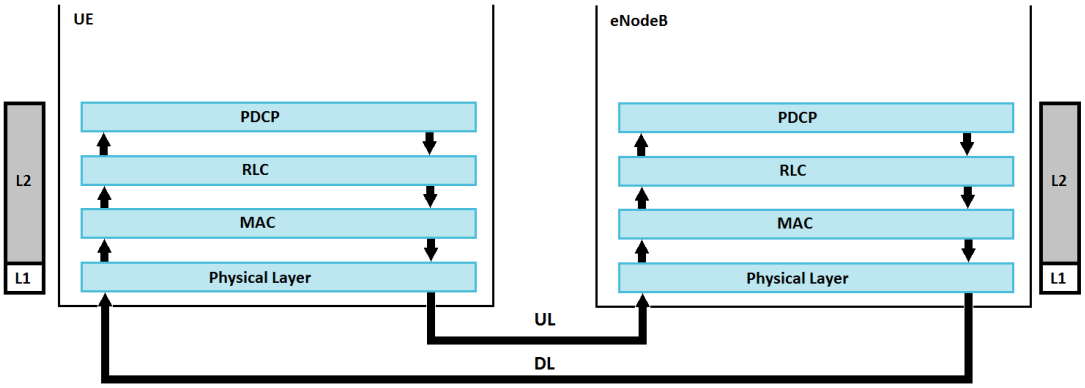

Figure 7. LTE User-Plane radio protocols and data flow between the eNodeB and the UE.

As can be seen from Figures 6 and 7, the LTE Layer 2 (L2) consists of three sub layers:

- Packet Data Convergence Control (PDCP)
- Radio Link Control (RLC)
- Medium Access Layer (MAC)

Figure 7 also shows the most important task of the LTE L2 sub layers, which is to transport data. Each sub layer naturally has additional tasks to perform in addition to transporting data but the figure illustrates the importance of performing these tasks quickly so that the data flow is as fast as possible. The faster the layers operate the faster the users can receive their data. Each layers' task in the eNodeB side is further explained in the following subsections.

2.3.1. PDCP

The PDCP layer has three main functionalities, which are header compression/decompression of IP packets, ciphering and deciphering and integrity protection and verification.

Header compression/decompression is based on the Robust Header Compression (ROHC) protocol. Efficient compression is important especially for small IP packets because if a large IP header is created, it will be a significant source of overhead for small data rates.

Ciphering and deciphering is done for both the User-Plane and the Control-Plane. Previously in 3G networks this was done in the MAC and RLC layers.

Integrity protection and verification is done for Control-Plane data and the purpose is to confirm that the control information is coming from the correct source. [1]

The PDCP layer's ROHC protocol is measured to be the major time critical algorithm, consuming approximately half of the entire L2 DL execution time based on measurements by D. Szczesny et al [9]. The measurements show that the computational power of the L2 DL is distributed as follows: PDCP 71 %, RLC 23 % and MAC 6 %.

2.3.2. RLC

The RLC layer transfers the protocol data units (PDUs) received from higher layers such as the PDCP to MAC layer and vice versa. The RLC also takes care of concatenation, segmentation, in-sequence delivery and error correction with automatic repeat request (ARQ). These actions are taken depending on the mode used.

There are 3 different modes in which the RLC can operate. They are Transparent Mode (TM), Unacknowledged Mode (UM) and Acknowledged Mode (AM).

Transparent Mode only relays the PDUs without adding any headers to them. This means that PDUs are not tracked. This mode is suited for services that don't require retransmissions and are not sensitive to delivery order.

Unacknowledged Mode includes in-sequence delivery of data, which comes in handy with data that has been received out of order due to Hybrid ARQ (HARQ) operation. On the transmitting side UM mode the data is segmented/concatenated to RLC Service Data Units (SDUs) with a UM data (UMD) header. The header includes the sequence number needed for in-sequence delivery. The formed data unit is called RLC PDU and is passed on to MAC layer.

Acknowledged Mode offers, in addition to the functionalities in UM mode, PDU retransmission if they are lost in lower layers. Information on the last correctly received packet is also stored and the header has the same sequence number as in UM mode. [1]

2.3.3. MAC

The MAC layer maps the logical channels used between the MAC and RLC layer to transport channels used between the MAC and the physical layer. On the downlink side the MAC layer does the multiplexing of RLC PDUs into Transport Blocks (TB) and delivers them to the physical layer on transport channels. The MAC layer operation on the uplink side is naturally the opposite of the downlink side operation. The MAC layer also takes care of error correction through HARQ, transport format selection and priority handling between logical channels. [1]

2.3.4. Cross-layer optimization

Performance wise it is not always beneficial to split layer functionalities to be completely separate. Cross-layer optimizations remove the strict boundaries between layers to allow communication between layers. This might include one layer to access the data of another layer to exchange information for the purpose of more

efficient performance. In one study cross-layer adaptation schemes are applied to the MAC and the physical layer, which results in improved average cell throughput by 25-60% [10].

2.4. LTE layers data flow

Figure 8 shows the data flow of the PDCP, RLC, MAC and the physical layer. Packets received by a layer are SDUs and the packets output of a layer are PDUs. The downlink data flow is explained in Subsection 2.4.1.

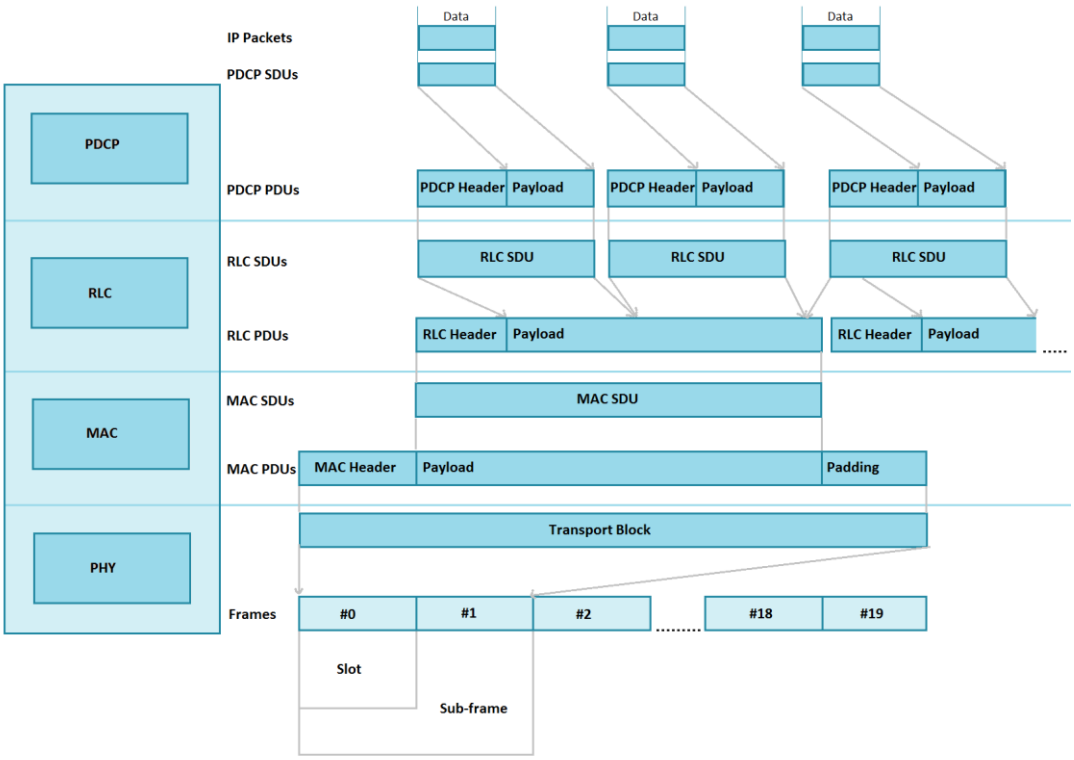

Figure 8. LTE data flow through different layers.

2.4.1. Downlink data flow

The downlink data flow starts from the IP layer, which sends packets to the PDCP layer. These packets are called PDCP SDUs. The PDCP layer then does header compression and adds a PDCP header to the PDCP SDUs. The header compression includes the removal of the IP header, which has the minimum size of 20 bytes, and adding a token, which is 1-4 bytes. Also ciphering for User and Control-Plane bearers is applied if configured. The data unit that is ciphered is the data part of the PDCP PDU. Then the PDCP layer submits the newly formed PDCP PDUs to the RLC layer.

The RLC layer receives the RLC SDUs (PDCP PDUs) and does segmentation on them. RLC segmentation might include splitting a large RLC SDU into multiple RLC PDUs or adding multiple small RLC SDUs into one RLC PDU depending on

the situation. After the segmentation is done the RLC PDUs are sent forward to the MAC layer.

The MAC layer receives the MAC SDUs (RLC PDUs) and then forms the MAC PDUs. A MAC PDU consists of the MAC header, MAC SDUs and MAC control elements. Then the MAC layer submits the MAC PDUs or Transport Blocks to the physical layer.

2.5. Focus of improvements

The improvement focus is on the software of the system rather than improving hardware or air-interface techniques. The goal is to make the LTE L2 functionalities to perform faster and to increase their capacity in users and radio bearers. In Chapter 3 many methods to boosting the performance of any embedded system by modifying its software are described.

3. PERFORMANCE OPTIMIZATIONS

Software optimization can be described as a process of modifying a software system to make some aspect of it work more efficiently or use fewer resources [11]. A program or a system can be optimized in many different ways. It can be optimized for example to:

- Execute code faster
- Operate with less memory
- Use memory more efficiently
- Draw less power

In this chapter different software optimization methods for boosting the performance of an embedded system, like the eNodeB, are introduced. The main focus area for this thesis work is to simply decrease the load of the central processing units (CPUs) executing the software of the LTE L2 User-Plane in an eNodeB and therefore increase the capacity of the network. This translates to making the code execute faster and to use the memory more efficiently.

3.1. Compiler

Compilers translate high-level programming languages such as C and C++ into assembly code for the target processor. In the early 2000 and before, most of embedded systems were programmed using assembly, and compilers were not widely used. Embedded systems have high-efficiency requirements, thus they require well optimized software [12]. Nowadays compilers are able to offer highly optimized code and there is less need for developers to program in assembly. Each processor has its own unique machine language. In optimization context this needs to be taken into account when choosing the compiler for an embedded system [13].

3.1.1. Compiler optimization options

Compilers provide optimization opportunities by different compilation flags. With GCC compiler turning on optimization flags makes the compiler attempt to improve the performance and/or code size at the expense of compilation time and possibly the ability to debug the system. [14]

Configuring and using different compiler flags can have a huge impact in performance as is shown in the book “Techniques for optimizing applications: high performance computing”. There different optimization flags for the Sun compiler and their impact in performance are described. One example shows a part of the code, which does matrix-matrix multiplication. The performance of the code improves more than eight times when using the “–x04” optimization level in the compiler compared to compilation with no optimization. [15]

Table 2 shows the differences between a couple different GCC optimization options. The “-O3”-option is widely used with GCC compilers if optimizations are needed. From the Table differences between the 3 levels of “-O”-options can be seen. For example using the “-O1”-option the compiler inlines functions that are called once with the flag “-finline-functions-called-once“, but with the “-O3”-option the

compiler tries to inline functions more aggressively with the flag "-finline-functions". Optimization instructions can also be given for the compiler in code with keywords and directives. The inline keyword along with others are explained in Section 3.3.

Table 2. GCC optimization options [14]

Option and description:	Enables:	
-O1 The compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.	-fauto-inc-dec -fbranch-count-reg -fcombine-stack-adjustments -fcompare-elim -fcprop-registers -fdce -fdefer-pop -fdelayed-branch -fdse -fforward-propagate -fguess-branch-probability -fif-conversion2 -fif-conversion -finline-functions-called-once -fipa-pure-const -fipa-profile -fipa-reference -fmerge-constants -fmove-loop-invariants	-fshrink-wrap -fsplit-wide-types -ftree-bit-ccp -ftree-ccp -fssa-phiopt -ftree-ch -ftree-copy-prop -ftree-copyrename -ftree-dce -ftree-dominator-opts -ftree-dse -ftree-forwprop -ftree-fre -ftree-phiprop -ftree-sink -ftree-slsr -ftree-sra -ftree-pta -ftree-ter -funit-at-a-time
-O2 -O2 turns on all the same flags as –O1 and also some additional flags. This option optimizes even more and also increases the compilation time as well as the performance of the generated code.	-fthread-jumps -falign-functions -falign-jumps -falign-loops -falign-labels -fcaller-saves -fcrossjumping -fcse-follow-jumps -fcse-skip-blocks -fdelete-null-pointer-checks -fdevirtualize -fdevirtualize-speculatively -fexpensive-optimizations -fgcse -fgcse-lm -fhoist-adjacent-loads -finline-small-functions -findirect-inlining -fipa-cp -fipa-sra -fipa-icf -fisolate-erroneous-paths-dereference	-flra-remat -foptimize-sibling-calls -foptimize-strlen -fpartial-inlining -fpeephole2 -freorder-blocks -freorder-blocks-and-partition -freorder-functions -frerun-cse-after-loop -fsched-interblock -fsched-spec -fschedule-insns -fschedule-insns2 -fstrict-aliasing -fstrict-overflow -ftree-builtin-call-dce -ftree-switch-conversion -ftree-tail-merge -ftree-pre -ftree-vrp -fipa-ra
-O3 -O3 turns on all optimizations in -O2 and adds a few more in an effort to optimize even more.	-finline-functions -funswitch-loops -fpredictive-commoning -fgcse-after-reload -ftree-loop-vectorize	-ftree-loop-distribute-patterns -ftree-slp-vectorize -fvect-cost-model -ftree-partial-pre -fipa-cp-clone

3.2. Dynamic approach

A dynamic approach means in the scope of this thesis, making performance optimizations dynamically. One of these dynamic optimizations is compiling code on the fly, which is called dynamic compilation. Dynamic compilation is usually good for code that executes over large sets of data using the same data processing loop. In addition dynamic compilation also results in a compact binary. In this section the concept of runtime code generation is explained along with dynamic compilers and their advantages and disadvantages.

3.2.1. Runtime code generation

Runtime code generation (RTCG) means dynamically adding code to the instruction stream of an executing program. Static-code implementation translates the code into an intermediate form, which is then interpreted with a statically-compiled interpreter. With RTCG implementation code is dynamically compiled, which means translating it to machine code and then executing it directly. RTCG is required for incremental compilers, dynamic linkers and debuggers. Some systems use RTCG to improve performance, such as interactive systems that demand good response time on inner loops.

Nowadays memories in embedded systems are large and fast memory is provided by caches. Fetching data fast is done by first-level caches, but if the data needed is not located in the cache, there is a large miss penalty, which has a significant effect on the performance of a system. Cache performance can be affected by implementation-dependant compiler optimizations, such as loop unrolling. RTCG lets a program dynamically adapt to these kinds of particular implementations. For example, if an array is known at runtime to be small, then the iterations can be fully unrolled. This is value-specific optimization (VSO), which means that when the program state and its inputs are known, the code can be optimized for specific values.

Runtime code generation is beneficial only when the generated code is run fast enough that it pays for the time spent optimizing. The payback depends on both the runtime cost of optimizing and how much faster and how many times the runtime-generated code executes. This is shown in Figure 9. [16] [17]

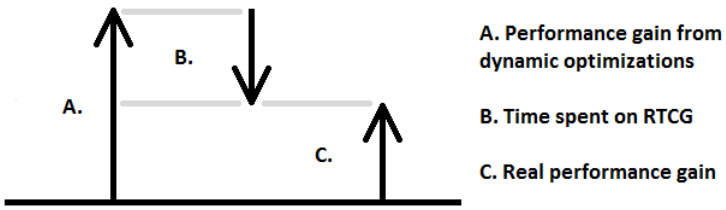

Figure 9. Visualization of real performance gain in dynamic optimizations.

3.2.2. Dynamic compilers

Compilation speed is crucial to RTCG's performance. Dynamic compilers or runtime compilers are usually faster than conventional compilers since they do not need to compile everything straight away. Some dynamic compilers are written for the particular application using statically generated templates, which contain "holes". The dynamic compiler then later fills those holes by the values computed at runtime.

With VSO the dynamic compiler can use all information available to a static compiler with the addition of the present information visible at runtime. This leads to a high possibility of generating better code than is possible with static compilation. [16] [17]

3.2.3. Advantages

Dynamic optimizations have advantages, such as faster execution time, reduced development cost and in some cases highly optimized code compared to code produced by other methods. The faster execution time with large data sets is demonstrated in Figure 10. Dynamic optimization reduces the development costs, since it lessens the need for optimizing by hand. Also in some cases hand-written optimizations in code have fallen short to dynamic optimizations made by the compiler. [16]

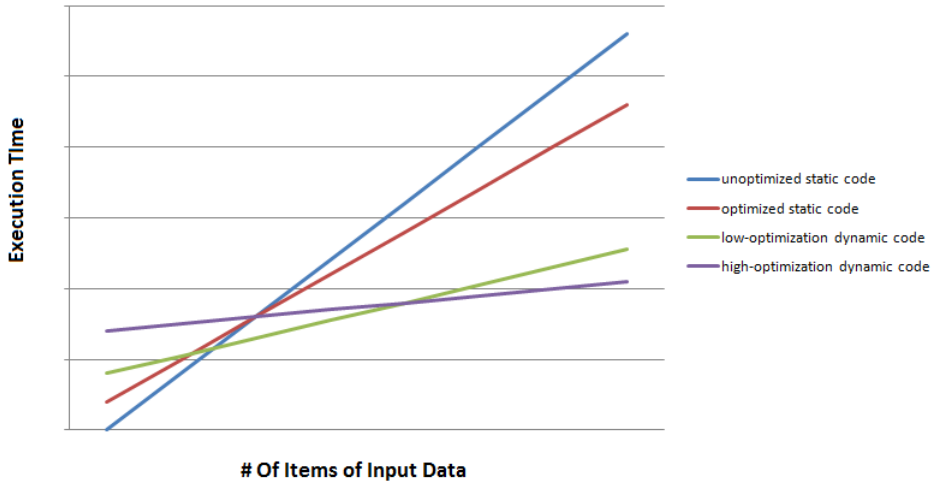

Figure 10. Execution times of dynamic and static optimizations.

3.2.4. Disadvantages

Disadvantages for dynamic optimizations are: platform dependency, added complexity and the need for big data sets to be profitable. Dynamic optimizations require a lot of work and modifications to work in a specific platform. This means that if the program is needed to run on many different platforms, dynamic optimizations need to be setup to work on all of them and the solutions might be distinctly different from each other.

Added complexity means making the code hard to read. With dynamic optimizations the code needs to be modified in certain places to make use of the optimizations. When the code is harder to read, it is harder to modify and therefore the further development is increasingly difficult.

As seen in Figure 10 dynamic optimizations are great for accessing or using large data sets in processing loops. But if the code is divided into small processing loops using different data sets within them, dynamic optimizations will not provide good results.

Dynamic optimizations were not further researched in this thesis. One reason for this was that the software of LTE L2 User-Plane in eNodeB is used with multiple platforms and the platform dependency of dynamic optimizations would not fit well. Another reason was that the code in the software is packet-specific, meaning that the execution for every packet is different and therefore hard to predict and divided into many processes.

3.3. Static approach

In comparison to the dynamic approach, a static approach means, within the scope of this thesis, making optimizations by hand rather than making them dynamically in runtime. A static approach was chosen to be used in performance optimizations in this thesis. This was because static optimizations guarantee performance enhancing results, if done correctly, compared to dynamic optimizations, which could have not worked at all.

Before starting any optimizations it is vital to identify the bottlenecks of the software. A bottleneck is a component or components that limit the entire performance of a system. In some programs 99% of the time is spent doing mathematical calculations, and in some that time is spent reading and writing data files, while using them takes less than 1% of the time. Optimizing less critical parts of the code is a waste of time and will make the code more difficult to debug and maintain. This is because optimizations reduce the readability of the code and make it more complex. [18]

This section shows how to find worthwhile optimization targets. Also useful and basic optimization methods are described. The programming language used in the software of the thesis work is C++, therefore most of the examples are written in it.

3.3.1. Code profiling

Profilers help in finding the bottlenecks or hotspots of a system. Profilers are tools that can for example tell how many times a function is called, how much time it uses and where the time is spent inside that function. Profilers are not without fault and it is usually best to use more than one method in finding the bottlenecks of the software.

There are many different profiling methods. Table 3 shows four different profiling methods with explanations on how the profiling works in the method. [18]

Table 3. Different profiling methods

Profiling method	What it does
Instrumentation	Adds instructions to the target program to collect the required information. That information might be how many times different functions are called.
Debugging	Inserts temporary debug breakpoints for example to every function or every code line.
Time-based sampling	Tells the operating system to create an interrupt for example every millisecond. With the information the profiler is able to count how many times the interrupts occur in different parts of the program.
Event-based sampling	Tells the CPU to generate interrupts at certain events. Events could be for example cache misses, data memory accesses and CPU cycles. This makes it possible to see which part of the program has most cache misses or CPU cycles and so on.

3.3.2. Compiler keywords

Compilers have many keywords and directives that are used for giving specific optimization instructions at specific places in code. Const and static are one of the most popular of these. They work on all C++ compilers and help the compiler to do optimizations. It is important for all software developers to understand how to use these keywords, because the more they are used the more the compiler can do additional optimizations.

Const tells that a variable is constant and will never change. This will allow the compiler to optimize the variable away in many cases. One example is shown in Code Fragment 1. There the compiler can replace all occurrences of ArraySize with the value 1000. This means that the compiler does not need to allocate any memory for ArraySize. [18]

Code Fragment 1. Const keyword example

```
const int ArraySize = 1000;
int Array[ArraySize];
…
for(int x = 0; x < ArraySize; x++)
{
   Array[x]++;
}
```

The meaning of a static keyword depends on the situation. In Table 4 different places where the keyword static can be applied and its impact are described.

Table 4. Static keyword descriptions

Applied to	Impact	Why
A non-member function	Tells the compiler that the function is not accessed by any other module.	Makes inlining more efficient and enables interprocedural optimizations.
A class member function	The function cannot access any non-static data members or member functions.	It is called faster than a non-static member function.
A global variable	Tells the compiler that the variable is not accessed by any other module.	Enables interprocedural optimizations.
A local const variable	Tells the compiler to initialize the variable only the first time the function is called.	Saves a lot of CPU cycles.

Interprocedural optimizations (IPO) refer to a collection of compiler optimizations or techniques used to improve performance in programs containing many frequently used functions. The difference between IPO and other compiler optimizations is that IPO analyzes the entire program, where other compiler optimizations look at smaller scope issues, such as a single function or a single block of code.

An example of applying the static keyword to a local const variable is displayed in Code Fragment 2. There function calculateVariable is called only once due to the usage of a static keyword. This means that a check must be added for the variable to see if the function has been called but execution is still faster than calling the function every time the exampleFunction is called. [18]

Code Fragment 2. Static keyword example

```
void exampleFunction()
{
...
  static const int exampleVariable = calculateVariable(1);
...
}
```

3.3.3. Integers, variables and operators

Integer operations are fast in most cases, regardless of the size. One thing to note is to avoid using integer sizes larger than the largest register size. For example, using 64-bit integers in 32-bit systems is inefficient.

Operations like addition, subtraction, comparison, bit operations and shift operations usually take only one clock cycle on most microprocessors. Multiplication and division are the most time consuming operations. Multiplication usually takes 4 cycles, whereas division might take up to 80 cycles, depending on the microprocessor.

It is usually more efficient to use bit shifting instead of multiplication or division, if possible. Bit shifting is possible if the values of the variables are to a power of 2. A

good optimizing compiler will replace multiplications with shifts, when possible, but programmers should make sure that this is done by the compiler especially in places where a lot of multiplication or division operations are executed. [18]

3.3.4. Functions

Excessive function calls may cause performance issues. The book "Optimizing software in C++" lists some of these. One of the major issues is that a lot of function calls lead the code to become fragmented and scattered in memory. This will require the CPU to jump to many different code addresses, which costs CPU cycles. There are a lot of solutions for these issues, but the most important are the following: avoiding unnecessary functions, using inline functions and avoiding nested function calls in the innermost loop.

Some coding conventions prefer that the size of a function should be only a couple lines. This is good when the functions do logically distinct tasks, but splitting functions to smaller ones should not be done only because the original function was too long. Also a critical innermost loop should always be kept inside one function, if possible. If a function has to be called inside a loop, inlining should be used.

The inline keyword is an extremely simple and powerful way for optimizing C++ programs. Inlining a function means that the place where the function is called, is replaced by the function body. The benefits from inlining are the performance gains coming from avoiding the function call, stack frame manipulation and the function return. The negative side is that inlining functions increase program size, build times and may increase the program's execution time by reducing the caller's locality of reference.

One example of the profits of inlining is described by Pete Isensee in "C++ Optimization Strategies and Techniques". There a situation is described where a C++ standard library sort function ran 7 times faster than one other C standard function qsort on a test of 500 000 elements because the C++ sort function was inlined.

Inlining a function is advantageous when the function is small or called from only one place. Good compilers also inline some functions automatically but in larger programs compilers usually do not inline all functions that could be inlined. This is why developers themselves should make sure that inlining is done by the compiler in places where it should be used. An example of inlining a function is shown in Tables 5 and 6. [18] [19]

Table 5. Before inlining example

Code	Explanation

ExampleClass.cpp
--- 	
void ExampleClass::exampleFunction_1(){
  ...
  setExampleVariable(5);
  ...
}
void ExampleClass::exampleFunction_2(){
  ...
exampleVariable=FncStr::getExampleVariable();
  ...
}
 	
FunctionStorage.hpp
--- 	
namespace FncStr{
  void setExampleVariable(int value){
  m_exampleVariable = value;
  }
  int getExampleVariable(){
  return m_exampleVariable;
  }
}
	Execution jumps to the addresses of setExampleVariable and getExampleVariable, when they are called.

Table 6. After inlining example

Code	Explanation

ExampleClass.cpp
--- 	
void ExampleClass::exampleFunction_1(){
  ...
  setExampleVariable(5);
  ...
}
void ExampleClass::exampleFunction_2(){
  ...
exampleVariable=FncStr::getExampleVariable();
  ...
}
 	
FunctionStorage.hpp
--- 	
namespace FncStr{
  inline void setExampleVariable(int value){
  m_exampleVariable = value;
  }
  inline int getExampleVariable(){
  return m_exampleVariable;
  }
}
	Compiler converts the code in functions exampleFunction_1 and exampleFunction_2 in the following way:
	
	ExampleClass.cpp
	--- 	
	void ExampleClass::exampleFunction_1(){
	  ...
	  m_exampleVariable = 5;
	  ...
	}
	
	void ExampleClass::exampleFunction_2(){
	  ...
	  exampleVariable= m_exampleVariable;
	  ...
	}

3.3.5. Branches

Modern microprocessors can achieve high performance partly due to using an instruction pipeline, where instructions are fetched and decoded in several stages before they are executed. In practice this means that the processor fetches a lot of instructions in advance before executing any of them. When the processor notices a branch, it has to decide, which of the paths it will feed to the pipeline. A branch might be a if-else, switch-case, for, while or do-while statement. If the processor feeds the instructions under the else-statement to the pipeline, when it should have fed the if-statement instructions, it can lose approximately 12-25 cycles, depending on the processor. This is called the branch misprediction penalty and the processor deciding which branch to feed to the pipeline, is called branch prediction. Branch prediction is done by algorithms that make their decision through past history of that branch and other nearby branches. [18]

Branches are usually cheap, when predicted correctly, but expensive if they are mispredicted often. Branches with a 50-50 chance of going either way will result in mispredictions 50% of the time. Naturally, this will have a negative impact on performance. Imperfect branch prediction can reduce performance by a factor of two to more than ten [20]. One simple example on how to avoid branch prediction misses is the binary lookup table, which is seen in Code Fragments 3 and 4. If the original code would cause a lot of mispredictions, it would be beneficial to use the look-up table since there the processor does not have to predict anything.

Code Fragment 3. Binary look-up table example, original code

```
int a; bool b;
…
if (b)
   a = 4;
else
   a = 2;
```

Code Fragment 4. Binary look-up table example, modified code

```
int a; bool b;
…
Static const type lookup_table[] = {2, 4};
a = lookup_table[b];
```

3.3.6. Error handling

C++ offers exception handling with try, catch and throw-methods in order to recover and handle errors. They are not very efficient performance-wise and should be avoided if possible. It is more efficient to define own error-handling methods, which simply print an error message and then try to exit or shut down the program or function. [18]

Error handling code is not executed very often but needs to be checked every time. This generates considerable amount of additional instructions to functions. In small scale this is not an issue, since jumping over error handling code does not have an impact on performance, if the program only has error handling in only a few parts of

the code. But in a larger scale error handling increases the size of the binary a great deal and generates a lot of inactive code.

One solution for optimizing error handling code is to move the error handling code to its own functions and make sure that it is not inlined. This is because the amount of generated instructions of an error handling code should be minimized in performance critical parts of the code.

3.3.7. Out-of-order execution

Out-of-order execution (OoOE) is an important term to understand for software developers, especially if their program is running on a microprocessor with OoOE support. If a processor has OoOE support, it executes instructions based on the availability of the input data, rather than their original order. With the help of out-of-order execution, the processor minimizes its idle time by not having to wait for the current instruction to finish before executing the second one. Code Fragment 5 and Table 7 show the advantage of out-of-order execution in a simplified example. There out-of-order execution completes the instructions 2 CPU cycles faster than in-order execution.

Code Fragment 5. Instructions of out-of-order execution example. Incrementing a variable takes one CPU cycle and loading a variable takes 4 CPU cycles

```
INCR A
LOAD B
INCR B
INCR C
INCR D
```

Table 7. Out-of-order execution example

Cycles	Out-of-order execution	In-order execution
1	INCR A	INCR A
2	LOAD B	LOAD B
3	INCR C	-
4	INCR D	-
5	-	-
6	INCR B	INCR B
7	-	INCR C
8	-	INCR D

One limit for OoOE is data dependency, which is also shown in the previous example. This means that for example, if the first instruction is to load variable A from memory and the second instruction is to increment that value, it cannot execute the second instruction before the first is completed.

3.3.8. Software pipelining

Software pipelining is defined as a coding technique that overlaps operations from various loop iterations in order to exploit instruction level parallelism (ILP) [21]. Software pipelining is a type of out-of-order execution, except that the reordering of

instructions is done by the compiler instead of the processor. ILP is a measure of how many operations can be performed simultaneously. If the developers want to make use of software pipelining, they must be aware of the architecture of the processor they are using. The important thing is to know how many instructions can be executed parallel and what kind of execution pipelines exist in the processor core.

For example, a fictional processor could dispatch 3 instructions parallel and have one load, one store, one simple logic, one multiply and one division execution pipeline. This practically means that it could execute simultaneously 3 operations, if the operations were different from each other. For example, it could do store, load, simple logic operation simultaneously, but could not do store, store, load simultaneously. In practice software pipelining is re-arranging code and the goal is to be able to execute instructions simultaneously as much as possible.

One example is shown in Code Fragment 6 and Table 8. There a loop is shown where in the first cycle one instruction is executed. In the second cycle two instructions are executed parallel and finally in third cycle three instructions are executed parallel to the point when there are only three instructions left in total. This is enabled by OoOE and software pipelining. The reason why three instructions cannot be executed at once from the start is the data dependency. Incrementing a variable before loading it is not possible.

Code Fragment 6. Software pipelining example

```
for (int i = 0; i < 100; i++)
{
  x = A[i];
  x++;
  A[i] = x;
}
```

Table 8. Instructions and their execution order of Code Fragment 6

CPU cycles	i = 0	i = 1	i = 2	i = 3
1	load A[0]	-	-	-
2	incr A[0]	load A[1]	-	-
3	**store A[0]**	**incr A[1]**	**load A[2]**	-
4		**store A[1]**	**incr A[2]**	**load A[3]**
5			**store A[2]**	**incr A[3]**
6				**store A[3]**

3.4. Memory optimizations

Memory plays an important role in the performance of embedded systems, such as the eNodeB, since the main task of the software of LTE L2 User-Plane is to transport data from the IP layers to the physical layer and vice versa. Therefore it is essential

that the software accesses its memory as fast as possible, since usually slow memory accesses are the most time consuming operations in embedded systems.

Nowadays with the new processors pushing the limits of high performance even further, the processor-memory gap widens and becomes the bottleneck in achieving high performance [22]. This section introduces CPU caches and their importance in performance optimizations. Also, different methods for optimizing the usage of CPU caches are discussed.

3.4.1. CPU caches

CPU caches try to bridge the gap between the CPU and the memory as they offer the illusion of a large and fast memory [23]. They are a much smaller memory and they try to keep the most frequently used data copied from the main memory. Usually CPUs have three independent caches as can be seen in Table 9.

Table 9. Cache types and their function

Cache type	Speeds up
Instruction cache	Executable instruction fetch
Data cache	Data store and fetch
Translation lookaside buffer (TLB)	Virtual-to-physical address translation

Data caches are usually split into a hierarchy of more cache levels. Figure 11 illustrates one of these hierarchies in a 4-core Shared Cache Chip Multiprocessor (CMP). It has on-chip core-specific L1 Instruction caches, L1 Data caches and a shared L2 Data cache between cores.

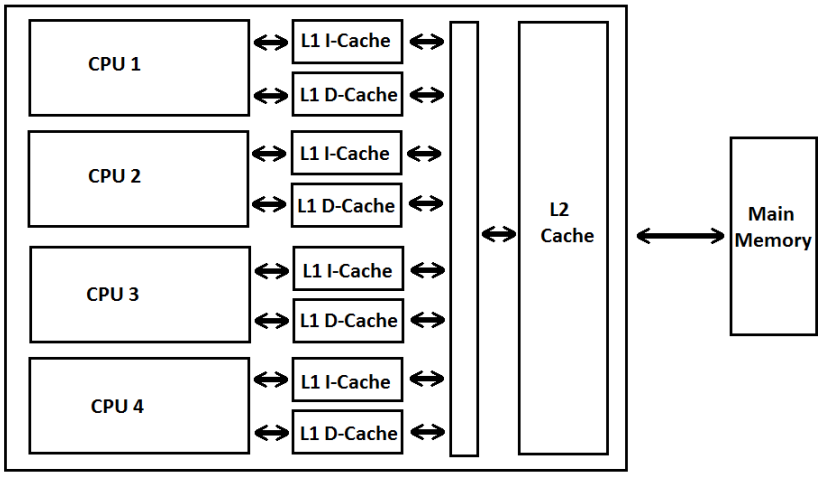

Figure 11. A 4-core CMP cache hierarchy.

3.4.2. Cache entry structure

The data copied from main memory is stored in a cache as cache lines (also known as cache rows). When a cache line is copied from memory into the cache, a cache entry is created. The usual structure of a cache entry is shown in Figure 12.

Figure 12. Cache entry structure.

Tag contains the address of the data fetched from the main memory. Data block contains the actual data. Flag bits indicate the state in which the cache line is. Data cache typically has two flag bits: a valid bit and a dirty bit. Valid bit indicates if the cache line is loaded with valid data. Usually valid bits are set to invalid in all cache lines when the hardware powers-up. This way the CPU knows when the cache lines are stale. Dirty bit, when set, indicates that the cache line has changed since it has been copied from main memory.

3.4.3. Cache entries

In read or write operations the processor first checks whether, the required data is located in the cache. If so, the processor performs the operation for the data located in the cache, if not the operation will be performed to the data located in the main memory, which is much slower. When the processor tries to handle data which is not located in cache, it is called a cache miss. When the data is found in the cache, it is called a cache hit.

In the case of a cache hit, the processor immediately reads or writes the data in the cache line. When a cache miss occurs, the cache allocates a new entry and copies the data from the main memory. The processor has to wait for the data to be ready in the cache to perform its read or write operation. CPU caches are a critical component of any high performance system and usually cache access time and cache misses are the single factor most constraining performance, since accessing main memory is much slower than accessing caches [24].

3.4.4. Cache replacement policy

Cache replacement policy determines which data stays in the cache and which is evicted. This is needed when cache misses happen and something is needed to be replaced to make room for the new entry. One of these policies is the least-recently used policy (LRU). It replaces the least recently accessed entry with the retrieved data. In one study different replacement policies are compared and their performance evaluated and it shows that there are substantial differences in performance of different replacement policies. [25]

3.4.5. CPU stalls

When a cache miss happens, the CPU will try to execute other instructions as much as possible, while fetching the cache line from memory. Usually though the CPU will run out of things to do before the fetching is complete. This is called a CPU stall and it is basically a moment where no instructions are executed and the CPU is idling. This of course should be avoided since it can cost hundreds of wasted CPU cycles. The penalty of a cache miss depends on how good the CPU is able to

hide it. The more the CPU can issue instructions during the cache miss, the better the penalty is hidden. [26]

The focus of the thesis in memory optimizations is mainly to try to avoid cache misses as much as possible. If avoiding a cache miss is not possible, then the penalty of the miss must be minimized. Techniques for avoiding cache misses and minimizing the penalty are described in the subsections to follow and in Section 3.5.

3.4.6. Code arrangement

Functions that are used near each other should be also stored near each other. The code cache benefits from this and works more efficiently. Also, it is efficient to collect functions that are used in the critical part of the code and store them together preferably in the same source file. Seldom used functions and code like error handling should be put to their own functions and be kept separate from often used functions.

Variables should be declared before they are needed inside functions and variables and objects like functions that are used close together should also be stored together. Static and global variables and dynamic memory allocation should also be avoided. [18]

3.4.7. Prefetching

Fetching a cache line that is expected to be used later is called prefetching. This is done to avoid cache misses that would happen without prefetching. Modern processors do prefetching automatically quite efficiently thanks to out-of-order execution and advanced prediction mechanisms. Prefetching is one of the several effective approaches for tolerating large memory latencies [18]. There are two main ways of prefetching: prefetching initiated by hardware and prefetching initiated by software.

Hardware prefetching is implemented by the processor. The implementation of the hardware prefetching is naturally different in different processors. Nowadays processors have multiple different hardware prefetchers, such as for example Intel Xeon® 5500 series processors, which have separate prefetchers for fetching data to L1 cache and for prefetching data to L2 caches. Hardware prefetchers can have different algorithms that decide which cache lines are fetched. One algorithm could always fetch two adjacent lines, where one could monitor data access patterns and try to predict from them the addresses needed in the future.

Software prefetching is done by the software developers manually. When doing software prefetches, the developers must have knowledge of the possible places where prefetching is needed. Different performance profilers can help in this by providing information on where, for example, CPU stalls happen. Doing software prefetching generally helps, but without performance profiling information it can harm the system's performance.

Prefetching has to be done in the correct time. Prefetching too early will result in replacing useful data or in worst cases the prefetched data will be replaced before being used. If prefetching is done too late, it will not be able to hide the processor stall. Figure 13 illustrates the effect of a successful prefetch on execution time. [27] [28]

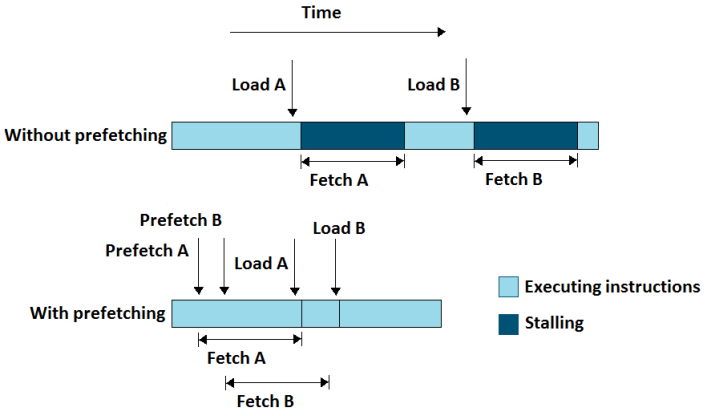

Figure 13. Prefetching example.

3.5. Loop optimization

The efficiency of a loop depends on the processor's ability to predict the loop control branch. An optimal loop, which is small and has a fixed repeat count with no branches inside, can be predicted perfectly. But in practice those loops are rare. [18]

Loop optimization is the process of optimizing loops, which results in faster execution time and less overhead. It boosts cache performance and uses parallel processing to its advantage. Loop optimizations are also very important since most of the execution time is spent inside loops. [29]

Extensive research has been done on loop optimizations and there are many different methods of doing them. The following subsections go through a couple of those methods that best fit into the theme for this research. One loop optimization that has been already described in Subsection 3.3.8 is software pipelining.

3.5.1. Loop unrolling

Loop unrolling is a technique that attempts to optimize a program's execution speed at the expense of binary size. It writes the loop or part of the loop open when used. Loop unrolling is helpful when a loop contains branches. It is able to remove those branches when done correctly and will also reduce the loop control branch significantly.

Code Fragments 7 and 8 show one example. Advantages of the loop unrolling are: loop control branch count reduced to 10 from 20 and the if-branch is removed completely. The disadvantage is that the unrolling creates more code to the binary. Overall the execution speed will increase by this change.

Code Fragment 7. Loop unrolling example, original code

```
for (int x = 0; x < 20; x++) {
   if (x % 2 == 0) {
      exampleFunction_A(x);
   }
   else {
      exampleFunction_B(x);
   }
   exampleFunction_C(x);
}
```

Code Fragment 8. Loop unrolling example, unrolled loop

```
for(int x = 0; x < 20; x+=2){
   exampleFunction_A(x);
   exampleFunction_B(x+1);
   exampleFunction_C(x);
   exampleFunction_C(x+1);
}
```

Some compilers like the GCC can unroll loops in the code automatically. In GCC this is done by enabling the flag "-funroll-loops". This option makes the code larger, but may or may not make the code run faster [14]. Loop unrolling should be done by the developer manually only if it appears beneficial, such as for example in the previous example, where the if-branch was eliminated. [18]

3.5.2. Loop fusion/fission

One obvious method for loop optimizations is the loop fusion. There two or more for-loops are combined into one, if the loops iterate over the same range and do not have data dependencies on each other. It might seem that loop fusion would always improve performance but this is not the case. In some cases two loops may perform better than one loop, which might be caused for example if the one loop contains an excessive number of computations. One example of loop fusion is illustrated in Code Fragments 9 and 10.

Code Fragment 9. Loop fusion example, original code

```
for (int x = 0; x < 100; x++){
   A[x] = A[x] + 5;
}
for (int x = 0; x < 100; x++){
   B[x] = B[x] + 10;
}
```

Code Fragment 10. Loop fusion example, combined loop

```
for (int x = 0; x < 100; x++){
   A[x] = A[x] + 5;
   B[x] = B[x] + 10;
}
```

3.6. Related work and results

Extensive research about software optimizations to increase performance in embedded systems has already been done over several decades. Usually optimizations go through the following pattern:

1. Software profiling
2. Algorithm optimization
3. Source code optimization
4. Memory optimization

In one embedded system all these methods were used and the results were performance increase of about 12%, memory access decrease of 72% and memory usage decrease of 35% over the original software. The source-code optimizations used were simple compiler optimizations and loop optimizations. Memory optimizations were made to optimize the usage of on-chip memory. [30]

In one other study optimizations were done for embedded software code running on ARM processors. There different loop optimization techniques were used along with other various optimization techniques and a 40% increase in the code density and 30% increase in execution time were achieved. Both of the studies used the static approach in optimizing and focused on making the usage of on-chip memory or caches efficient as they usually are the biggest obstacle in achieving better performance. [31]

Dynamic optimizations have also yielded good results. In one study dynamic partitioning was used for software and hardware and the program executed 2.6 times faster. Dynamic partitioning meant monitoring CPU's executing binary program, detecting critical code regions, decompiling those regions, synthesizing them to hardware, placing and routing that hardware onto on-chip configurable logic and updating the binary to communicate with said logic. [32]

In another study software execution was accelerated by using modulo scheduling heuristic to map applications into a virtual coarse-grained reconfigurable architecture. The success of this method depends on the success of extracting loops and mapping them into the architecture. The study presented a runtime solution, which showed gain of 3 to 6 times, when comparing compilation times. The solution enabled the use of dynamic compilation, which made it adaptive to changing scenarios. [33]

The merits of both static and dynamic optimizations have been researched and presented in this chapter. Both of the optimization methods can outperform each other and the decision on which of the optimization methods should be used, depends on the situation. In this thesis only static optimizations are used and in the next chapter the performance evaluation framework and tools for verifying the optimization work are described.

4. PERFORMANCE EVALUATION FRAMEWORK AND TOOLS

The main and most important task for the software of LTE L2 User-Plane is to transport data from the IP layer to the physical layer and vice versa. The details of the different layers of L2 are explained in Chapter 2. Figure 14 illustrates this further with a simplified view of the data flow through LTE L2 User-Plane. In this chapter details about the software's test environment are described with the focus on performance evaluation.

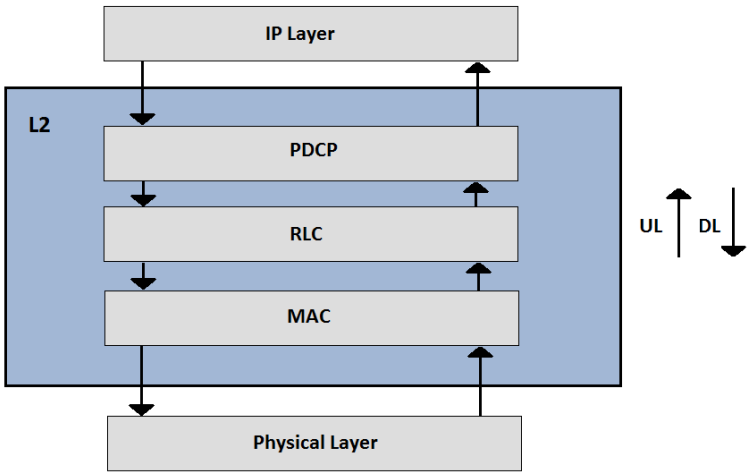

Figure 14. UL and DL data flow through L2 User-Plane.

4.1. Test environment

There are many different levels of testing that are needed to ensure that the software of LTE L2 User-Plane is working as intended. There are unit tests, module tests, component tests and system component tests to name a few. The purpose of, for example unit testing is to test the smallest testable parts of the software. This usually means running the tests locally and making sure that the functions and classes work as intended.

System component tests (SCT) verify the software's functionalities in the real eNB hardware. One SCT suite is for example *L2 Cell Setup*, which ensures that everything related to cell setups is working correctly in L2. For this reason in system component testing a couple of simulators are needed. MAC Scheduler, IP layer data generator and physical layer simulators act as the real boundaries that would communicate with the software of LTE L2 User-Plane in the real eNB. The performance testing is done with SCT.

The SCTs use *Robot framework* as the test automation framework and continuous integration is handled by an open source continuous integration tool called *Jenkins*. Performance profiling is done with *perf*-tool or by monitoring performance counters directly. [34] [35] [36]

4.1.1. Robot framework

Robot framework is a generic test automation framework for acceptance testing. It has its own test data syntax and utilizes the keyword-driven testing approach. The core framework is implemented using Python and testing capabilities can be extended by adding test libraries implemented with Python or Java. [34]

The advantages of *Robot framework* are its easily approachable syntax and the extensive logfiles that it creates. The test cases should be written in a way that the name describes the test's intent and the test case itself should be easily understandable. In Code Fragment 11 one example is shown of a test case written in *Robot*.

Code Fragment 11. *Robot* example, test code

```
*** Settings ***
Test Setup        Start CPU load measurements      Preconditions of cell setup
Test Teardown  Kernel error check in teardown    Verify default CPU loads    Postconditions of
cell setup

*** Variables ***
${CELL ID}=           45

*** Test cases ***
Successful cell setup
   Successful cell setup to LTEL2  ${CELL ID}

*** Keywords ***
Successful cell setup to LTEL2
   [arguments]  ${cell_id}
   Successful MAC cell setup LTEL2   ${cell_id}
   Successful TUP cell setup   ${cell_id}
```

Robot also easily enables pre- and postconditions or setups and teardowns to tests as can be seen from the Code Fragment. This means that if multiple tests were added to the test suite, they would automatically run the test setup and teardown that are written under **** Settings ****.

In Figure 15 a part of the generated log.html file is shown. It shows the test case *Successful cell setup* along with the keywords that the test has. If one of the keywords should fail in the execution, the test case logfile would indicate the exact keyword where the fail happened along with the reason that caused the fail.

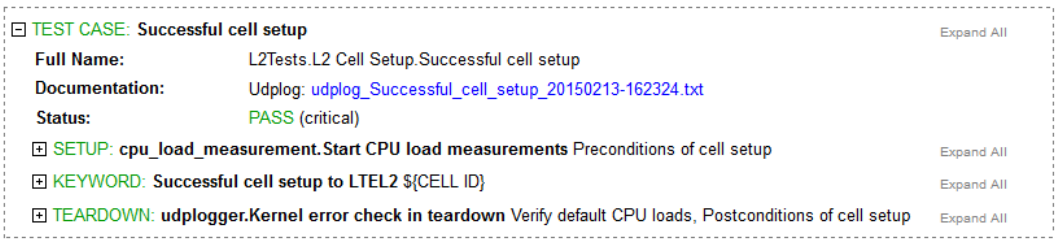

Figure 15. *Robot* example, part of logfile.

4.1.2. Continuous integration

As mentioned previously the software's continuous integration is handled by the application *Jenkins*. *Jenkins* runs jobs that can include different builds and test lists. The jobs are initiated by software configuration management (SCM) changes. This means that when a developer uploads his or her changes into the version control system, *Jenkins* notices this and initiates the jobs. If some of the jobs fail, the person who committed the changes will be notified by an automatic email. Every job has its own page, where its history can be looked through. The job-specific pages can show for example the generated logfiles of the tests. [35]

Jenkins also makes it easy to monitor the effect in performance, when the code is updated. In the performance profiling jobs a database and a graphical view of the results was updated after every run. This is shown in Figure 16. The y-axis indicates the average CPU load and the x-axis shows the revision number, which is a specific id given by the version control system to every change done to the software.

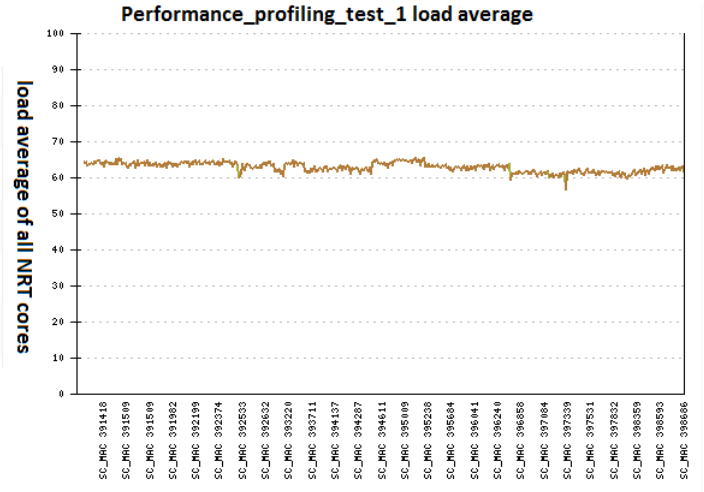

Figure 16. Performance profiling, whole system's CPU load graph.

The database in question also stored function-specific loads, which were accessible through *Jenkins*. Figure 17 shows a function-specific CPU load view and the effect of optimizations done to the function.

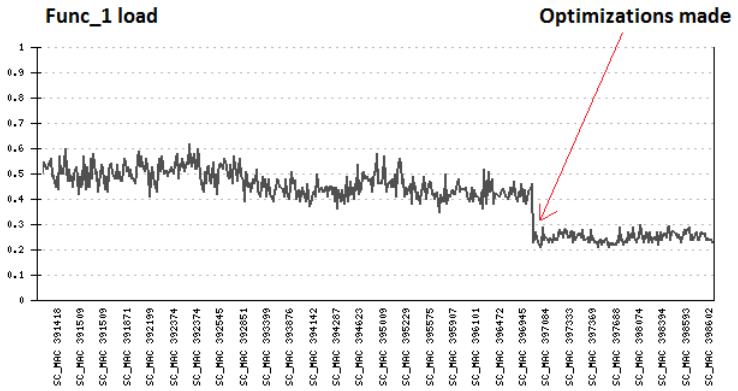

Figure 17. Performance profiling, function-specific CPU load graph.

4.2. Performance profiling with perf

Perf is the performance profiling tool used in the thesis work. It is available for Linux and supports hardware performance counters, tracepoints, software performance counters and dynamic probes. Performance counters are CPU hardware registers that count hardware events such as cache-misses, instructions or branch mispredictions. Tracking these registers makes it easy for *perf* to identify the hotspots of the system. *Perf* also has the ability to drill down to the source code and assembly level of the functions or methods in the code and show the statistical distribution of where the time is spent in that specific function. [36]

4.2.1. Recording samples

Perf can collect performance profiles on per-thread, per-process and per-cpu basis. *Perf* uses event-based sampling, which is described in Subsection 3.3.1. By default *perf* uses cycles as the sampling event. *Perf* records samples at certain intervals, which can be configured by the user.

For example, if the sample rate is 50 000, this means that *perf* records a sample after the event has occurred 50 000 times. If the recorded event is L1 cache-misses and the sample rate 1000, this means that after 1000 L1 cache-misses *perf* records a sample. The recorded sample is the line in code where the process was when the sample was recorded. *Perf* then attaches the sample to the function or method in which the code line was located.

The performance profile created by *perf* is based on statistical probability. Thus the collected sample count should be high enough for *perf* to create an accurate profile. If the amount of overall samples is low, the results of the measurement can be considered unreliable. Choosing the correct sample rate depends greatly on the event measured. If cycles are measured, then the sample rate can be set quite high, since the overall sample count with cycles is high. If L2 cache-misses are recorded, then the sample rate should be quite low, since L2 cache-miss events do not happen that often.

The *perf record* command can be given for example in the following way: *perf record -o perf.data -C 0-3 -c 50000 -e cycles sleep 2*. This is further explained in Table 10.

Table 10. Example *perf record* command

Command type (*perf*)	**Output filename (-o)**	**CPU/CPUs (-C)**	**Sample rate (-c)**	**Event (-e)**	**Measurement time (sleep)**
record	perf.data	0-3	50000	cycles	2

4.2.2. Sample analysis

The samples that *perf record* collects are stored in the output binary file specified in the *record* command. In the example in Table 10 the samples would have been saved into file *perf.data*. These samples can then be analyzed with *perf report* by

giving the following command: *perf report --show-nr-samples -i perf.data*. *Perf report* reads the *perf.data* file and generates a concise execution profile. A part of an example profile is shown in Table 11.

Table 11. *Perf report* example profile

Percentage of overall samples	Number of samples	Process	Library	Function
41.01 %	229978	Disp_1	libA.so	Func_1
2.16 %	12093	Disp_1	libB.so	Func_2
2.10%	11790	Disp_1	libC.so	Func_3

Perf report also provides a user interface, which can be used to browse the performance profile created by *perf record*. With the user interface it is easy to, for example, zoom into only functions located in *libA.so*.

It is also possible to save the performance profile with *perf report* to a different file. In LTE L2 User-Plane *perf report* is used to generate a textfile of the performance profile, which is later used to update the performance profile database described in Subsection 4.1.2.

4.2.3. Detailed analysis

Perf can also drill down to the assembly level with *perf annotate* to show the statistical distribution of samples within a function. *Annotate* shows the real bottlenecks and hotspots, since it shows where the execution was most of the time. Using *perf annotate* through the interface provided by *perf report* is also very simple. For example, in one case a performance profile was created by recording CPU cycles. *Perf annotate* was then used on one of the functions with the highest sample counts. Code Fragment 12 shows a part of that view. From there it can be interpreted that loading *rb_ctx->transferActive* takes 23.47% of the function's time.

Code Fragment 12. *Perf annotate* example

```
        inline RlcSupportUlRbContext_t * getRb(TRbIndex rbIndex)
        {
          if(rbIndex < MAX_NUM_OF_RB_PER_UEGROUP) {
            RlcSupportUlRbContext_t * rb_ctx = rbCtx + rbIndex;
              return rb_ctx;
 0.48     ldr.w  ip, [sp, #96]   ; 0x60
          add.w  r0, r6, r6, lsl #5
          add.w  r0, ip, r0, lsl #7
           if(L2LMgrAPI::isRbSetMAC(rbIndex, ueGroup))
           {
            RlcSupportUlRbContext_t * rb_ctx = q_ctx->getRb(rbIndex);      /* RB exists */
            if(rb_ctx->transferActive)
           add.w  r2, r0, #4160   ; 0x1040
23.47       ldr    r2, [r2, #28]
        cmp    r2,
```

4.2.4. System component testing with perf

The test environment in LTE L2 User-Plane SCT is setup in a way where the tests are run in the target hardware and the tests are only issued to start from the developers' work environment. Figure 18 presents an activity diagram on how the *perf* functionality is added into running the performance tests.

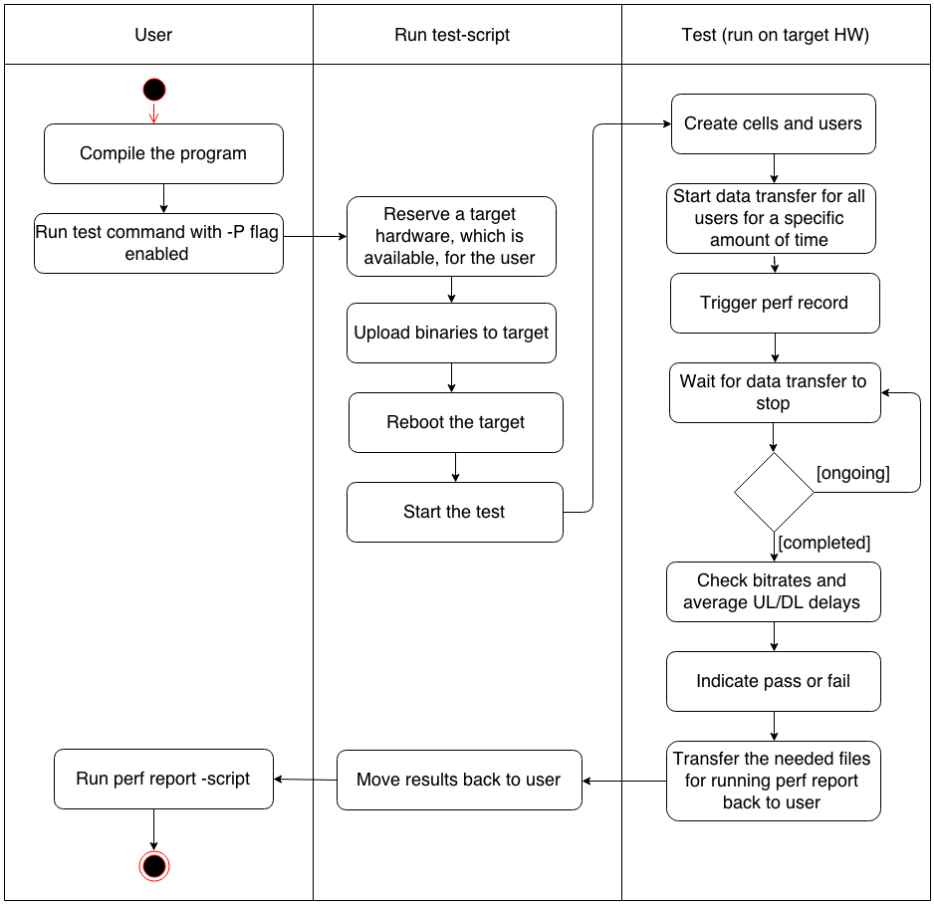

Figure 18. Activity diagram of how the test environment operates with *perf*.

In short the *run_test*-script is told with a *-P*-flag that *perf* measurements should be initiated. This enables the *perf* related functionalities added into the performance tests, which are: starting *perf record* when data transfer is ongoing for the specified amount of users and transferring the needed files back to the developer's work environment. This way the developer can run *perf* with only two commands: starting test with *perf* and running *perf report*. Customization options were also added, which are further described in Table 12.

Table 12. Customization options with *perf*

Command:	Added:	Defaults to:
run_test_script -P -p '--test test_case_1'	This is the default test command with *perf* enabled.	-
run_test_script -P -p '--test test_case_1 –v SAMPLERATE:20000'	-v SAMPLERATE:20000 Records with sample rate of 20 000	50 000
run_test_script -P -p '--test test_case_1 –v CPUS:1'	-v CPUS:1 Records CPU 1.	0-3
run_test_script -P -p '--test test_case_1 –v PERFTIME:30'	–v PERFTIME:30 Records for 30 seconds.	5
run_test_script -P -p '--test test_case_1 –v EVENTS:cache-misses’	–v EVENTS:cache-misses Records cache-misses.	cycles

4.3. Performance profiling manually

The performance profiles created by *perf* are good for showing the hotspots and bottlenecks of the software. However it was noticed during the optimization work that the profits of smaller scale optimizations were barely or not at all visible with *perf*. The solution was to track the performance counters manually.

A few methods were added to do the tracking. The two most important methods of these are *startPerformanceCounters()* and *readPerformanceCounters()*. Usage of these methods is shown in Code Fragment 13.

Code Fragment 13. Tracking performance counters manually example

```
startPerformanceCounters ();
function_under_inspection();
readPerformanceCounters ();
```

The function *startPerformanceCounters()* resets the values of performance counters and the function *readPerformanceCounters()* prints a new row into the logfile, which shows the current values of the counters. Table 13 and Code Fragment 14 show the counters as they are shown in the logfile along with the descriptions.

Code Fragment 14. Fragment of the logfile, when performance counters are tracked manually

```
VIP/cycles: 296 inst:  95 bpMiss: 0 memAcc:  71 L1Fill:  1 L2Fill:  0 uaAccess:  0  i/c:0.32
VIP/cycles: 296 inst:  95 bpMiss: 0 memAcc:  71 L1Fill:  0 L2Fill:  0 uaAccess:  0  i/c:0.32
VIP/cycles: 305 inst:  97 bpMiss:0 memAcc:   71 L1Fill:  3 L2Fill:  0 uaAccess:  0  i/c:0.32
```

Table 13. Abbreviations and descriptions of the counters printed into the logfile

Abbreviation	Amount of
cycles	CPU cycles
inst	Instructions
bpMiss	Branch prediction misses
memAcc	Memory accesses
L1Fill	L1 cache misses (When L1 cache is filled, it means a L1 cache miss has occurred)
L2Fill	L2 cache misses (When L2 cache is filled, it means a L2 cache miss has occurred)
uaAccess	Unaligned accesses

The value for *i/c* is calculated by dividing instructions by cycles. It is a measure of how efficient the profiled code is. If a CPU is able to process 3 instructions in one CPU cycle, then it is theoretically possible to achieve the *i/c*-ratio of 3. Therefore if that CPU would have the ratio of 0.32 shown in Code Fragment 14, conclusion could be made that the profiled code is not very efficient.

4.3.1. Accurate tracking of performance counters

Tracking the performance counters manually would result in the logfiles filling up of rows shown in Code Fragment 14. It is hard to interpret the effects of optimizations to the profiled code if the results show over one hundred thousand rows of results each with varying values. A script was made to count the overall and average values of the performance counters from the logfiles. In practice the performance counters were tracked in the following way:

1. Set the function *startPerformanceCounters()* before and the function *readPerformanceCounters()* after the code to be profiled
2. Compile and run the test
3. Run the *overall_average_counts*-script

The average counts made the results much more reliable and easier to follow. Table 14 shows the average counts of performance counters of one profiled function before and after optimizations provided by the script.

Table 14. Average counts of performance counters of one profiled function before and after optimizations

	cycles	inst	bpMiss	memAcc	L1Fill	L2Fill	uaAccess	i/c
Before optimizations	655	153	0	79	6	1	0	0.2336
After optimizations	514	134	0	74	4	1	0	0.2614

4.4. Performance profiling tests

The tests used in performance profiling are based on the requirements received from the customer. The requirements are then bundled together when specifying the performance test cases, for the purpose of verifying the performance in a few test

cases rather than making a new test for every performance requirement. In addition the performance test cases try to verify the most used and the most important functionalities.

The tests are specified with different targets. One test could target to have 4 cells, 1000 users per cell with 500 users with data with varying data rates. After running the test it could be then noticed that it fails, for example because the system cannot handle 500 users with data at once. A test case activity variable was created to handle this issue. It controls how many users have data. For example, if the test case activity was 25%, this would mean that there would be 125 users with data. In general with performance tests the test case activity variable is set to the last value that the test passes with. This way the system's performance is always known and the test cases can be quickly modified to increase the activity percentage closer to the target of 100%, when the performance of the system has improved.

During the thesis work a couple of test cases were selected for performance profiling. The test case activity with these tests was low in the beginning of the optimization work, but as time passed and new optimizations were introduced, the activity was raised.

5. PERFORMANCE IMPROVEMENTS

The optimization work included using many different optimization methods and this chapter shows a few examples of them. The work was done with the help of a team focused on optimizations. The *perf* results were recorded by monitoring CPU cycles and in some cases cache misses. In all optimizations also performance counters were monitored manually to back up the results provided by *perf*. This chapter does not include examples of all optimization methods used, since some optimization efforts did not produce worthwhile results.

5.1. Functions

It was noticed during the optimization work that there were a lot of places in the code where functions were called either unnecessarily or too many times. These situations force the CPU to execute obsolete code and should always be eliminated from the software. This section shows a couple of examples where function-specific optimizations were used.

5.1.1. Unnecessary function calls

Unnecessarily called functions do not execute any meaningful code. One example of an unnecessary function call is shown in Code Fragment 15. There two functions are called excessively since they are a part of the process of building MAC PDU. The function *addControlElements* calls the function *buildControlElements*, where code is located in three different if-clauses. It was noticed during the experimentation that the code in the latter function was cold-code, which means that it was barely executed. All the three if-clauses were checked but very rarely were true and the function was not even inlined, which caused an additional performance penalty.

Code Fragment 15. Unnecessary function calls, before optimizations example

```
inline void addControlElements()
{
      .... code here ....
     buildControlElements(taCeInfo, drxCommEnable, getUeCaCeInfo(p_CW));
}

void buildControlElements(const STaCeInfo      &p_taCeInfo,const
EDrxCommEnable &p_drxCommEnable, const SCaCeInfo      &p_caCeInfo)
{
   if( p_taCeInfo.taCeAvail )
   {
     .... code here ....
   }
   if( EDrxCommEnable_On == p_drxCommEnable )
   {
     .... code here ....
   }
   if ( p_caCeInfo.caCeAvail )
   {
     .... code here ....
   }
}
```

The most beneficial solution was to check the if-clauses already in *addControlElements* and therefore in most cases block the call for the function *buildControlElements*. Code Fragment 16 shows the optimized version of the function *addControlElements*.

Code Fragment 16. Unnecessary function calls, after optimizations example

```
inline void addControlElements(const TTbSize p_tbSize, const u32 p_CW)
{
      .... code here ....
     if (taCeInfo.taCeAvail ||
        (EDrxCommEnable_On == drxCommEnable) ||
        caCeInfo.caCeAvail)
     {
        buildControlElements(taCeInfo, drxCommEnable, caCeInfo);
     }
}
```

The effect of the optimization was minor and only a 2.5% load decrease was seen with *perf*, but since the optimization was small, the most reliable results could be received by measuring performance counters manually. Table 15 shows the results of the measurement and shows a 6.5% decrease in CPU cycles. Although the gains were not very large, they display a good practice that was followed in other optimizations as well.

Table 15. Average counts of performance counters when measuring functions *addControlElements* and *buildControlElements*

	cycles	**inst**	**bpMiss**	**memAcc**	**L1Fill**	**L2Fill**	**uaAccess**	**i/c**
Before optimizations	296	95	0	71	1	0	0	0.32
After optimizations	277	70	0	50	0	0	0	0.25

5.1.2. Excessive function calls

Perf also exposed a couple of simple functions that should be only called once at initialization phase, but in the current implementation were called excessively. These function calls caused unnecessary load and one example is shown in Code Fragment 17. There a function call of *AaConfigRadGetValue* was executed every time when a RLC PDU was handled in AM mode. The code was only run in the SCT environment, but caused a significant load in the performance profile, which lead to the need to optimize it.

Code Fragment 17. Excessive function calls, before optimizations example

```
TBoolean IsPduAccepted()
{
   .... code here ....

   if(AaConfigRadGetValue( (u32) ERadSwDomainLteMac_TRlcTesterDlCapacityTest) == 1 )
   {
      maxUlTbSizeInBytes = MAX_DL_TB_SIZE_IN_BYTES;
   }

   .... code here ....
}
```

The solution was to add a couple of static boolean variables to the class, which can be seen in Code Fragment 18. The variables *m_isAaConfigRadGetValueCalled* and *m_rlcCapacityTestsOn* were set to false on initialization and later used as is shown in the Code Fragment. In practice they made sure that the function *AaConfigRadGetValue* was called only once and kept the value returned from the function in the variable *m_rlcCapacityTestsOn*.

Code Fragment 18. Excessive function calls, after optimizations example

```
TBoolean IsPduAccepted()
{
   .... code here ....

   if(!m_isAaConfigRadGetValueCalled)
   {
      m_isAaConfigRadGetValueCalled = true;

      if( AaConfigRadGetValue( (u32)ERadSwDomainLteMac_TRlcTesterDlCapacityTest) == 1 )
      {
         m_rlcCapacityTestsOn = true;
         maxUlTbSizeInBytes = MAX_DL_TB_SIZE_IN_BYTES;
      }
   }
   else if (m_rlcCapacityTestsOn)
   {
      maxUlTbSizeInBytes = MAX_DL_TB_SIZE_IN_BYTES;
   }

   .... code here ....
}
```

The same kinds of optimizations were introduced to three different classes and the performance effect was quite large. Before optimizations the function load of *AaConfigRadGetValue* was 8041 samples and after 4445 samples, when measuring with *perf*. This results in a load decrease of 44.72% for the function. The effect of the optimization in the scope of the whole software is shown Figure 19.

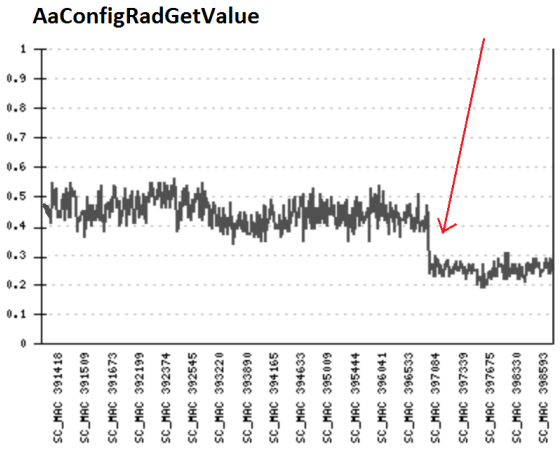

Figure 19. Optimization impact on *AaConfigRadGetValue*.

5.2. Branches

Branches have a negative impact on the performance of a system, when they are predicted poorly by the microprocessor. Code Fragment 19 displays a part of code, which generated branch mispredictions. In the Code Fragment a function for fetching a context for tunnel endpoint id is presented. The function was called only from one function *EmLocalTeidLookupTable::find(),* which had a significant performance impact on the system.

Code Fragment 19. Branch prediction misses, before optimizations example

```
static inline const L2LManager_teidCtx_t* getLocalTeidCtxByTeid(const TGtpTeid teid)
{
   .... code here ....

   while ((tmp_index != invalidteidIndex) &&
          (getLocalTeidCtx(tmp_index).teid != teid))
   {
      tmp_index = getLocalTeidCtx(tmp_index).next;
   }

   if (tmp_index != invalidteidIndex)
   {
      return &getLocalTeidCtx(tmp_index);
   }

   return GLO_NULL;
}
```

Through experimentation it was noticed that most of the time the first while-loop did not iterate at all, since *getLocalTeidCtx(tmp_index).teid* equaled the variable *teid* with a 90% chance in the first iteration. This resulted in the usage of a common path first optimization approach, which is organizing the code so that the common path executes first. Code Fragment 20 presents this solution.

Code Fragment 20. Branch prediction misses, after optimizations example

```
static inline const L2LManager_teidCtx_t* getLocalTeidCtxByTeid(const TGtpTeid teid)
{
   .... code here ....

   while(tmp_index != invalidteidIndex)
   {
      if(getLocalTeidCtx(tmp_index).teid == teid)
      {
         return &getLocalTeidCtx(tmp_index);
      }

      tmp_index = getLocalTeidCtx(tmp_index).next;
   }
   return GLO_NULL;
}
```

As can be seen from the Code Fragment 20, the code now executes the path when *getLocalTeidCtx(tmp_index).teid* and *teid* are equal first. Measuring performance counters manually shows that the CPU cycles of the function drop from 622 to 524 on average, which is decrease of 15.76%.

Table 16. Average and overall counts of performance counters when measuring function *EmLocalTeidLookupTable::find*

	cycles	**inst**	**bpMiss**	**i/c**
Average counts				
Before optimizations	622	135	1	0.2182
After optimizations	524	135	0	0.2580
Overall counts				
Before optimizations Samples: 169 926	105 807 932	23 058 655	241 831	0.2179
After optimizations Samples: 169 923	89 100 814	22 969 447	90 874	0.2578

From Table 16 it can also be calculated that there is a 62.42% drop in branch prediction misses, which has the biggest impact in reducing CPU cycles in this case. Organizing code common path first is a good practice and was also utilized in other optimizations.

5.3. Error handling

The software of LTE L2 User-Plane has a lot of error handling code. This is because the software can come across many different fault scenarios when run in the field. Therefore it is important to be able to track what went wrong in the software, and the error handlers have been implemented to address that issue.

Performance-wise error handlers have a negative impact, since error handlers force the code to do a check which is obsolete most of the time. Error handlers also bloat the binary by adding a lot of cold-code in the middle of the active code and therefore force the code to jump a lot in its execution. There is also a big risk of getting cache misses from instruction caches, if the code has an overwhelming amount of error handling instructions. Instruction cache misses cause the most delay out of other type

of cache misses because the processor, or at least the thread of execution, has to wait until the instruction is fetched from main memory. An example of an error handler is shown in Code Fragment 21.

Code Fragment 21. Error handlers before optimizations example

```
void macPduBuilderPtrgenerateMsg(CDlMacPduBuilder *m_macPduBuilderPtr,
                                 const LogConnectionId logCid)
{
  switch(m_macPduBuilderPtr->getMsgType())
  {
     ... code here ...
    default:
    {
      LOG_LTEMAC_ERR1(FID_DLDATA, "RlcCoreDl", logCid,
        "PduMuxBundledDataReq: Unknown reqType:%d received in SPduMuxDataReq",
        (i32)m_macPduBuilderPtr->getMsgType() );
      REPORT_FAULT_WITH_FAULTREPORTER( EFaultId_UnknownResourceIdAl
          , L2LMgrAPI::getLomFaultReporter());
      break;
    }
  }
}
```

The solution for decreasing the negative impact of error handlers was to separate them into their own functions and cold-sections in the binary. This was implemented to all RLC DL core component functionalities, since those functionalities can be considered as the hottest parts of the software of LTE L2 User-Plane. Code Fragment 22 displays the optimized version of Code Fragment 21.

Code Fragment 22. Error handlers after optimizations example

```
void macPduBuilderPtrgenerateMsg(CDlMacPduBuilder *m_macPduBuilderPtr,
                                 const LogConnectionId logCid)
{
  switch(m_macPduBuilderPtr->getMsgType())
  {
     ... code here ...
    default:
    {
      handleUnknownReqTypeError(logCid);
      break;
    }
  }
}
SECTION (".ColdCode") )
void handleUnknownReqTypeError(const TLogConnectionId logCid)
{
  LOG_LTEMAC_ERR1(FID_DLDATA, "RlcCoreDl", logCid,
    "PduMuxBundledDataReq: Unknown reqType:%d received in SPduMuxDataReq",
    (i32)m_macPduBuilderPtr->getMsgType() );
  REPORT_FAULT_WITH_FAULTREPORTER( EFaultId_UnknownResourceIdAl
      , L2LMgrAPI::getLomFaultReporter());
}
```

The size of the active binary in RLC DL core reduced from 55 000 bytes to 48 810 bytes, which is a decrease of 11.25%. The performance counters were manually measured by monitoring one of heaviest functions in RLC DL core. The CPU cycle count was on average 22 085 CPU cycles before optimizations and 20 430 after. This meant that after implementing the error handlers, the functions in RLC DL core executed 7.5% less CPU cycles. Table 17 presents the values of performance counters when instruction cache related counters were monitored.

Table 17. Average counts of instruction cache related performance counters

	cycles	**inst**	**L1iAccess**	**L1iRefill**	**i/c**
Before optimizations	22085	9820	4318	361	0.4447
After optimizations	20430	9876	4157	289	0.4834

The instruction cache misses were reduced by 19.94% and accesses by 3.73%. This lead to the conclusion of separating cold-code results into a more efficient code. The positive impact discovered in these results concluded in the usage of separating error handlers into their own methods in other optimization work as well.

5.4. Memory optimizations

CPU stalls can be considered as one of the biggest reasons for a system to perform poorly. CPU stalls are usually caused by cache misses, since the processor needs to fetch the data from main memory and will eventually run out of things to do while loading the data. During optimization work cache misses were identified as the biggest individual factors for poor performance inside badly performing functions. This section focuses on showing a couple of examples on how to minimize cache misses and therefore reduce CPU stalls.

5.4.1. Prefetching

With the help of *perf annotate* it was quite easy to find out the code and functions that caused the most CPU stalls. In Code Fragment 23 the function *processPduMuxDataCombReqs* is shown, which was one of the high load functions of the software of LTE L2 User-Plane. The Code Fragment shows that the highest load in the function is caused by the line *m_cfi = msgPtr->cfi*. This was because *pduMuxDataCombReq*-struct was not located in cache on most iterations of the for-loop and needed to be fetched there from the main memory.

Code Fragment 23. Prefetching example before optimizations with *perf annotate*

```
void processPduMuxDataCombReqs()
{
    ... code here ...
   for(u32 i=0; i < numOfPduMuxReqs; i++)
   {
     initPduMuxDataReq( &pduMuxDataCombReq[i]);
     ... code here ...
   }
}

inline void initPduMuxDataReq(SPduMuxDataCombReq * msgPtr)
{
   ... code here ...
   m_cfi = msgPtr->cfi;
27.03       ldrb   r0, [r4, #8]
   ... code here ...
}
```

The solution was to prefetch the next *pduMuxDataCombReq[i]*-struct one iteration before it was needed. Prefetching does not always work due to needing the data either too early or too late. In this case the CPU could easily fetch the needed data before the next iteration, since it had an adequate amount of code to be executed before the next iteration. The first iteration could not be prefetched, as the function call for *processPduMuxDataCombReqs* could not be predicted early enough to have enough time to initiate and complete the prefetch. As can be seen from the Code Fragment 24, the load from the *m_cfi = msgPtr->cfi*-line has decreased from 27.03% to 4.32%.

Code Fragment 24. Prefetching example after optimizations with *perf annotate*

```
void processPduMuxDataCombReqs()
{
   ... code here ...
   for(u32 i=0; i < numOfPduMuxReqs; i++)
   {
     initPduMuxDataReq( &pduMuxDataCombReq[i] );

     if(i+1 < numOfPduMuxReqs)
     {
       DATA_PREFETCH_2(&pduMuxDataCombReq[i+1]);
 0.17       pld    [sl, #292]      ; 0x124
 0.03       pld    [r9]
     }
     ... code here ...
   }
}

inline void initPduMuxDataReq(SPduMuxDataCombReq * msgPtr)
{
   ... code here ...
   m_cfi = msgPtr->cfi;
4.32       ldrb.w r0, [sl, #8]
   ... code here ...
}
```

Perf recorded 18 383 samples from the function before optimizations and 13 288 samples after, which means that the function's load decreased by 27.72%. Although prefetching provided great results in this case, it could not be used in all cache miss situations, since for example some functions, which had a lot of cache misses, could not be predicted early enough for the prefetch to complete.

5.4.2. Bit-table

Code Fragment 25 shows another example of a cache miss. There in function *fillLostBSRs* the variable *rb_ctx->newDataOctets* was checked for every radiobearer in a short period of time. Every radiobearer had its own context where the variable *rb_ctx->newDataOctets* was located and the size of that context was over 4000 bytes.

The CPU had to jump through big data sets in a very short period of time and usually, because of the large size of the context, the data was not located in cache, which caused a significant penalty in performance. Also during experimentation it was noticed that the value of *rb_ctx->newDataOctets* was zero over 95% of the time. If the value of *rb_ctx->newDataOctets* was zero, the function did not have to execute any meaningful code and was wasting a lot of time fetching the radiobearer contexts for nothing. Prefetching also was not an option since the for-loop iterated 37 times and the loop itself did not have enough code that could be arranged so that prefetching could be executed in time.

Code Fragment 25. Bit-table example before optimizations with *perf annotate*

```
void fillLostBSRs(const TCellIndex cellIndex) const
{
    ... code here ...
   u16 lastHandledRbIndex = q_ctx->getLastHandledRbIndex(cellIndex);

   for(u32 i = 0; i < NUM_OF_RBS_TO_CHECK_FOR_LOST_BSRS_PER_TTI; i++)
   {
       ... code here ...
       RlcSupportDlRbContext_t* const rb_ctx = q_ctx->getRb(lastHandledRbIndex);
       if(rb_ctx->newDataOctets != 0)
77.87      ldr.w  r3, [r8, #140]  ; 0x8c
       {
          if(rb_ctx->bsrTimeStamp != INVALID_VALUE_U16 &&
            rb_ctx->numOfBsrResends < MAX_NUM_OF_BSR_RESENDS)
            {
0.05        ldrh.w r3, [r8, #1194] ; 0x4aa
            .... code here ....
            }
        }
   }
}
```

The solution was to create a bit-table *m_rbsHavingOctets* that was the size of the amount of radiobearers. The bit-table stored the status of the variable *rb_ctx->newDataOctets* for every radiobearer and was updated every time the value *rb_ctx->newDataOctets* was updated. The idea behind the bit-table is presented in Figure

20. It demonstrates how transforming the needed data from a big data set into a much smaller compressed data set helps in accessing memory.

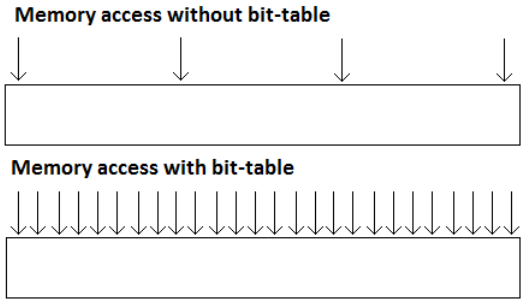

Figure 20. Idea behind the bit-table.

The function before optimizations iterated 37 times in the for-loop and then stored the last handled radiobearer id, so that it could continue next time from the point it left off. After optimizations *CBitGranularArrayIterator*-class was used to perform this functionality. The bit-table, number of bits to iterate and the starting bit index were passed on to the class when it was created. Then the function *bitArrayIterator.hasNext()* checked if any of the bits on the iteration range were 1, if that was the case then it would execute the code in the while-loop and save the radiobearer id to variable *rbIndex* as is shown in Code Fragment 26. In practice this meant that, because the value for *rb_ctx->newDataOctets* was zero over 95% of the time, the code inside the while-loop was executed only 5% of the time, whereas before the optimizations every iteration of the for-loop was executed.

Code Fragment 26. Bit-table example after optimizations with *perf annotate*

```
void fillLostBSRs(const TCellIndex cellIndex) const
{
    ... code here ...
   u16 lastHandledRbIndex = q_ctx->getLastHandledRbIndex(cellIndex);

   CBitGranularArrayIterator bitArrayIterator(
         q_ctx->getRbsHavingOctetsTable(),                          // bit table
         NUM_OF_RBS_TO_CHECK_FOR_LOST_BSRS_PER_TTI,   // num of bits to iterate
         lastHandledRbIndex);                                          // starting bit index

   while (bitArrayIterator.hasNext())
   {
      const u32 rbIndex = bitArrayIterator.next();

      ... code here ...
      RlcSupportDlRbContext_t* const rb_ctx = q_ctx->getRb(rbIndex);
      if(rb_ctx->bsrTimeStamp != INVALID_VALUE_U16 &&
         rb_ctx->numOfBsrResends < MAX_NUM_OF_BSR_RESENDS)
       {
1.17       ldrh.w r2, [r9, #1194] ; 0x4aa
            .... code here ....
       }
   }
}
```

The performance effect measured with *perf* is shown in Table 18. The samples of only the function *fillLostBSRs* and *fillLostBSRs* and the functions that needed to update the bit-table were measured. The table shows that the sample difference after optimizations was 4169 samples, when measuring all the functions affected by the optimizations. When this value is divided by 6020, which is the before optimizations sample count of *fillLostBSRs*, the real load decrease of the optimization can be calculated as 69.25% in the scope of the *fillLostBSRs* function.

Table 18. *Perf* sample counts of bit-table optimizations

	Only *fillLostBSRs*	**Function *fillLostBSRs* and functions that update *rb_ctx->newDataOctets***
Before optimizations	6020	17502
After optimizations	376	13333
Sample difference	5644	4169

The drop in cache misses was also confirmed by the results of manually monitoring performance counters, which are shown in Table 19. There the most important column is *L2Fill*, which shows a decrease of 99.45% in L2 cache misses. Naturally every other measured counter shows a significant decrease in them as well. This can be explained by two reasons: the new while-loop arrangement blocks most of the unnecessary execution of the code and accessing the *m_rbsHavingOctets*-table does not produce cache misses.

Table 19. Average and overall counts of performance counters when measuring function *fillLostBSRs*

	cycles	**inst**	**memAcc**	**L1Fill**	**L2Fill**
Average counts					
Before optimizations	2176	975	263	20	12
After optimizations	616	312	99	3	0
Overall counts					
Before optimizations samples: 4 383 275	9 490 171 966	4 297 989 117	1 154 438 764	89 960 680	53 087 419
After optimizations samples: 4 507 876	2 777 150 682	1 409 267 224	448 862 761	17 600 409	290 538

During optimization work it was noticed that optimizing cache misses had to be done case by case. The reason for cache misses needed to be analyzed and usually a very specific solution for optimizing them was the most efficient. The same solution very rarely worked for a different cache miss situation.

5.5. Loop optimizations

Most time in any software is spent in loops; therefore optimizing loops usually provides good results. Code Fragment 27 shows an example of a badly performing loop. The function was detected to perform poorly with *perf* after one set of measurements was switched on in the performance profiling test cases.

Code Fragment 27. Loop fusion example before optimizations

```
void UpdateValuesIfResultsAboveThreshold (const TCellId cellId, u32 const logChIndex, const u32
prevTime, const u32 newTime, const u32 numOfMeas)
{
   for( u32 i = 0; i < numOfMeas; i++ )
   {
      const u8 measIndex =  m_measMapping[ i ];

      if( cellId == m_measValues[ measIndex ].cellId )
      {
         const Threshold oldResult = CheckThreshold( logChIndex, prevTime, cellId );
         const Threshold newResult = CheckThreshold( logChIndex, newTime, cellId );

         .... code here ....
      }
   }
}

Threshold CheckThreshold(u32 const logChIndex, u32 const time, TCellId const cellId) const
{
  Threshold result = INVALID_PARAMETER;

  for (u32 cellIndex  =  0; cellIndex < MAX_CELLS_L2; cellIndex++)
  {
    if (cellId == m_measInfo[cellIndex].cellId)
    {
      u32 const threshold  =  m_measInfo[ cellIndex ].MeasInfo[logChIndex]. threshold  ;

      result = (time >= threshold  ) ? TH_EXCEEDED : TH_NOT_EXCEEDED;
    }
  }
  return result;
}
```

Function *CheckThreshold* was called twice in a row and the function itself executed a for-loop, where it would calculate if a time threshold had been exceeded for the argument *time*. It was clear that the functionality of the two consecutive function calls could be fused into one and check the time threshold for both arguments *prevTime* and *newTime* simultaneously. This way one of the for-loops could be eliminated altogether.

It was also noticed that the function call of *CheckThreshold* was inside the main for-loop inside *UpdateValuesIfResultsAboveThreshold* with no real reason, since all the arguments the *CheckThreshold* needed were already resolved before the main for-loop. By moving *CheckThreshold* before the main for-loop many obsolete calls for *CheckThreshold* were eliminated.

Also during experimentation it was noticed that the variables *oldResult* and *newResult* were *TH_NOT_EXCEEDED* most of the time. Further experimentation

revealed that the loop did not execute any meaningful code if that was the case. This would mean that the whole main for-loop could be blocked most of the time. Code Fragment 28 shows the optimized version of the code.

Code Fragment 28. Loop fusion example after optimizations

```
void UpdateValuesIfResultsAboveThreshold (const TCellId cellId, u32 const p_logChIndex, const u32
prevTime, const u32 newTime, const u32 numOfMeas)
{
      Threshold oldResult = INVALID_PARAMETER;
      Threshold newResult = INVALID_PARAMETER;

      if(IsUpdateNeeded(p_logChIndex, prevTime, newTime, cellId, oldResult, newResult))
      {
         for( u32 i = 0; i < numOfMeas; i++ )
         {
            const u8 measIndex = m_measMapping[ i ];

            if( cellId == m_measValues[ measIndex ].cellId )
            {
               .... code here ....
            }
         }
      }
}

inline TBoolean IsUpdateNeeded (const u32 logChIndex, const u32 prevTime, const u32 newTime,
const TCellId cellId, Threshold& oldResult, Threshold& newResult)
{
   bool updateNeeded = true;

   for( u32 cellIndex = 0; cellIndex < MAX_CELLS_L2; cellIndex++ )
   {
      if( cellId == m_measInfo[ cellIndex ].cellId )
      {
         const u32 threshold  = m_measInfo[ cellIndex ].MeasInfo[ logChIndex]. threshold  ;

         oldResult = (prevTime >= threshold  ) ? TH_EXCEEDED : TH_NOT_EXCEEDED;
         newResult = (newTime >= threshold  ) ? TH_EXCEEDED : TH_NOT_EXCEEDED;

         if((oldResult == TH_NOT_EXCEEDED) && (newResult == TH_NOT_EXCEEDED))
         {
            updateNeeded = false;
          }
      }
   }
   return updateNeeded;
}
```

The results of the optimizations were measured with *perf* in two phases. First only loop fusion of *CheckThreshold* was measured. Secondly in addition to the loop fusion, moving the fused function out of the main for-loop and blocking the main for-loop if both *oldResult* and *newResult* were *TH_NOT_EXCEEDED,* was measured.

Table 20. *Perf* sample counts of *UpdateValuesIfResultsAboveThreshold* with different optimizations

Optimizations:	Samples	Decrease (%)
None	5610	-
Only loop fusion of *CheckThreshhold*	4232	24.56
All	1610	71.30

Table 20 shows a decrease of 71.30% in the CPU cycles caused by the function *UpdateValuesIfResultsAboveThreshold* and this is also visualized in Figure 21, where the load drop can be seen in the scope of the whole software. The load drop can be easily explained by the fact that the function has to execute much less code now and even the whole main for-loop is blocked most of the time.

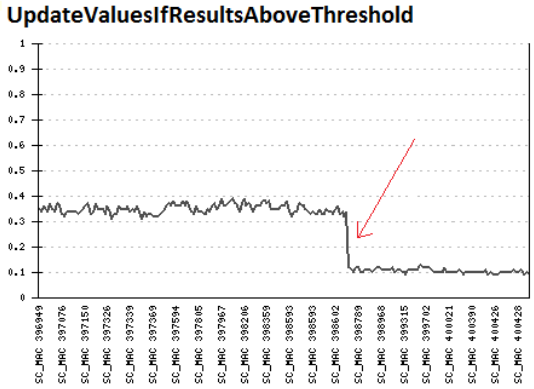

Figure 21. Optimization impact on *UpdateValuesIfResultsAboveThreshold*.

5.6. Overall impact of optimizations

A couple of different capacity and performance test cases were used in the thesis work as explained in Section 4.4. The results presented thus far have all been in the scope of a specific function, but the optimization impact on the whole software needed to be verified. One way of doing this was done by observing the maximum test case activities, in which the tests passed. Figure 22 displays the changes in the maximum activity in two different test cases used in performance testing.

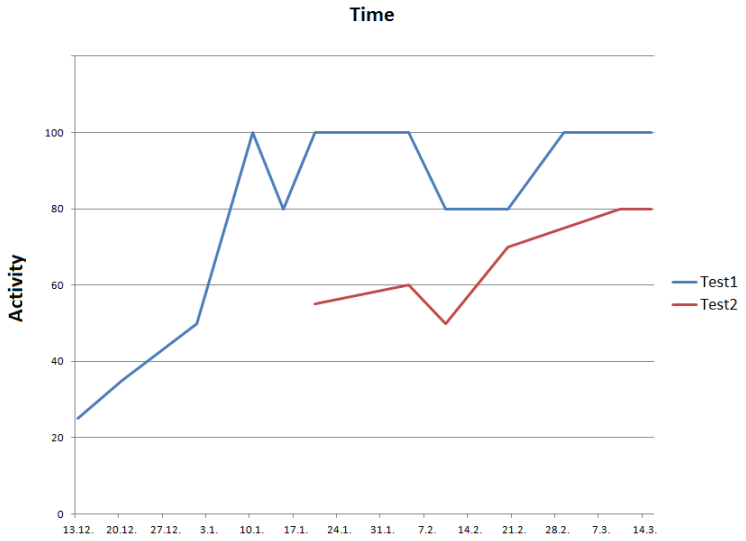

Figure 22. Changes in test case activities.

The figure shows the positive trend in activity over time. At the beginning of the optimization work only *Test1* was used for the overall performance measurements, but after it achieved 100% activity *Test2* was introduced with more demanding requirements. The drops in activity can be explained by the fact that over time many different features were included in the testing. For example, in the early February, various measurements were activated in the tests and the drop in activity caused by this can be seen in the figure.

Another way of measuring the optimization impact in the whole software was done with *perf*. *Perf* measured the idle processes in its performance profiles, which were then subtracted from the other results and then presented the remaining value as load. Figure 23 shows the overall load change measured in *Test2*.

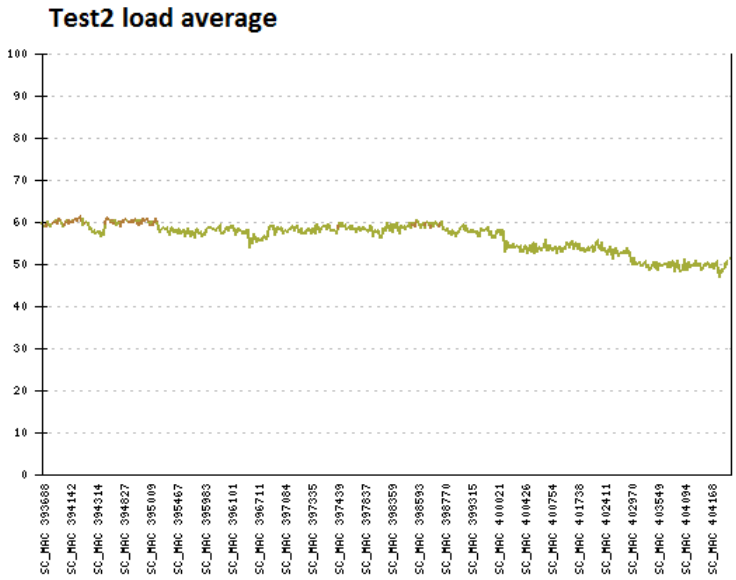

Figure 23. Overall load average of test 2.

The figure shows a drop from 60% load to 50% load, which can be interpreted as a significantly positive optimization impact. The optimization work also included configuring the most optimal compilation flags and optimizing the usage of different methods provided by other interfaces in addition to everything described in this chapter. Although the results of the optimizations have been quite positive, still work remains to achieve the activity of 100% in *Test2*.

6. DISCUSSION

The goal of the thesis was to optimize the software of LTE L2 User-Plane in eNodeB. The most important optimization targets were to increase the capacity of the system and decrease the processing latencies between layers and therefore decrease the latency UEs are experiencing. The most impactful optimizations used were either memory optimizations or loop optimizations, which were similar to the static optimization work described in papers seen in Subsection 3.6 [30] [31].

The positive impact of memory optimizations can be explained by the nature of the L2 User-Plane software, which is mostly transporting data. This means that the software needs to constantly go through large data sets in short periods of time, which creates high need for optimal memory access. The positive impact of loop optimizations stems from the fact that most of the time in any software is spent inside loops and the software of L2 User-Plane is no exception. Also other optimization practices were used, such as sectioning cold code into its own sections, arranging code common path first or reducing the amount of unnecessary function calls.

Perf performance profiler was essential during optimization work, since it pinpointed the hotspots in the code and gave immediate feedback on the optimization work's impact. Manually observing performance counters proved to be also a very successful way in verifying optimization results, since one could easily see the reason why specific optimizations worked the way they did.

The overall optimization results showed a decrease from 60% to 50% in the system's CPU load. Also test case activities for the most important capacity tests nearly achieved the targets set for them. The increase in test case activities translates to having significantly more users with ongoing data transfer between them. Therefore the results can be interpreted as very positive in both reducing CPU load and adding capacity to the system. However, optimization work needs to continue to reach the 100% activity in the capacity tests.

During optimization work different optimization methods were discovered for future work that could have had a positive impact on performance. One issue in the L2 User-Plane software was too large data structures and contexts. One example of this is shown in Subsection 5.4.2 where before optimizations data context of over 4000 bytes for every radiobearer was accessed in a short period of time. The solution was the bit-table, which avoids the access of the context entirely, but reducing the size of the original context was not investigated. The context could have been easily reduced in size for example by moving the cold parts of the context into their own structures and simply adding a pointer to the new structures to the original context. This could have lead to the needed data being located in cache when accessing it.

Sectioning code into cold and hot-sections was done in the RLC DL core component functionalities and could have been done for the whole software to see its optimization impact system-wide. It was analyzed that the impact would be certainly positive but the extent remained unclear, since sectioning the whole software would have taken too much time to implement within the timetable of the thesis.

Organizing code to take use of software pipelining was analyzed but not used, since the code consisted of mainly store, load and compare operations due to the nature of the software. This was also seen when monitoring code manually with performance counters in the instruction-cycle ratio. The ratio showed values in the range of 0.3-0.8 in different parts of the code, when the theoretical maximum of the

used processor was 3. One could argue that the code is not very optimal due to this fact, but when the software uses only a couple of different instructions, it is difficult to make use of software pipelining.

Overall the optimization team felt that the optimization impact of the thesis work was satisfactory. The optimization work is now easy to continue with the help of the information provided by the thesis work and with the knowledge on how to use the tools needed for performance profiling.

7. SUMMARY

The main objective of the thesis work was to increase the capacity of the software LTE L2 User-Plane in eNodeB. The LTE L2 User-Plane radio protocols are complex and their implementation depends greatly on the software, which leaves much room for software optimizations. Thus the optimizations in the software of LTE L2 User-Plane will have a significant positive impact on the performance of the whole eNodeB.

Two approaches in optimizations were introduced; static and dynamic. Dynamic approach is shown to be good for software which executes over large sets of data using the same data processing loop, but bad for software, where execution is hard to predict. Therefore a static approach was chosen for the optimizations, which is optimizing by hand instead of optimizing the code at runtime as in the dynamic approach. The static approach used in the thesis work focused on the most effective optimization practices in embedded systems, such as memory optimizations and loop optimizations, but also other common practices are discussed, such as using compiler keywords, minimizing branch prediction misses and making use of software pipelining.

The study used two methods for verifying the performance effect of software optimizations: *perf* and manually observing performance counters. *Perf* is a performance profiler tool which makes it easy to identify the hotspots of the system with function-level accuracy. *Perf* also gives the option to drill down to the code and see the actual lines which are causing the performance issues inside a function. Manually observing performance counters provided by the processor gives the confirmation on why specific optimizations worked, since they are able to monitor the amount of, for example, cycles, instructions, branch prediction misses and L1- or L2-level cache misses.

The most effective optimizations were found to be memory optimizations and loop optimizations. Memory optimizations were effective because the software needed to constantly go through large data sets and in many parts of the software the needed data was not located in the data caches, therefore accessing the data was causing CPU stalls. Loop optimizations on the other hand were effective, since the original loops before the optimizations included questionable choices in their code structure. Also other good common optimization practices were used, such as organizing code common path first, sectioning cold code into their own sections and blocking unnecessary function calls.

The optimization results showed a 70% load decrease in function-level at best and a decrease from 60% to 50% in the system's load after thesis work. Also the system's capacity was observed with the test case activities in the most important capacity test cases. The activities show a significant growth in their values, but still work remains to achieve the 100% activity in all important capacity test cases.

8. REFERENCES

[1] H. Holma and A. Toskala, LTE for UMTS OFDMA and SC-FDMA Based Radio Access, Chichester: John Wiley & Sons Ltd., 2009.

[2] E. Dahlman, S. Parkvall, J. Sköld and P. Beming, 3G Evolution HSPA and LTE for Mobile Broadband, Oxford: Academic Press, 2008.

[3] J. Huang, F. Qian, A. Gerber, O. Spatscheck, S. Sen and Z. M. Mao, "A Close Examination of Performance and Power Characteristics of 4G LTE Networks," ACM, Lake District, 2012.

[4] LteWorld, "LteWorld," [Online]. Available: http://lteworld.org/wiki/lte-advanced. [Accessed 10 2014].

[5] Cisco, "Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2013–2018," 2014. [Online]. Available: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white_paper_c11-520862.pdf.

[6] S. Gunelius, "Newstex, Mobile Devices to Surpass the Number of People on Earth – Infographic," 3 5 2014. [Online]. Available: http://newstex.com/wp-content/uploads/2014/05/the-golden-age-of-mobile-infographic.png.

[7] F. Firmin, "3GPP," [Online]. Available: http://www.3gpp.org/technologies/keywords-acronyms/100-the-evolved-packet-core.

[8] Freescale Semiconductor, "Freescale," [Online]. Available: http://www.freescale.com/files/wireless_comm/doc/white_paper/LTEPTCLOVWWP.pdf. [Accessed 10 2014].

[9] D. Szczesny, A. Showk, S. Hessel and A. Bilgic, "Performance Analysis of LTE Protocol Processing on an ARM based Mobile Platform," in *System-on-Chip, 2009. SOC 2009. International Symposium on*, Tampere, 2009.

[10] T. Kwon, H. Lee, S. Choi, J. Kim, D.-H. Cho, S. Cho, S. Yun, W.-H. Park and K. Kim, "Design and implementation of a simulator based on a cross-layer protocol between MAC and PHY layers in a WiBro Compatible.IEEE 802.16e OFDMA system," *Communications Magazine, IEEE,* vol. 43, no. 12, pp. 136-146, 2005.

[11] R. Sedgewick and K. Wayne, "Algorithms," Addison-Wesley Professional, 1984, p. 84.

[12] R. Leupers, "Compiler design issues for embedded processors," *Design & Test of Computers, IEEE,* vol. 19, no. 4, pp. 51-58, 2002.

[13] M. Barr, Programming Embedded Systems in C and C++, O'Reilly, 1999.

[14] "GNU compilers," Free Software Foundation, Inc. , 1988-2015. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#Optimize-Options.

[15] R. P. Garg and I. Sharapov, "Techniques for optimizing applications: high performance computing," 2001, pp. 127-130.

[16] D. Keppel, S. J. Eggers and R. R. Henry, A Case for Runtime Code Generation, Washington: University of Washington Department of Computer Science and Engineering, 1991.

[17] D. Keppel, S. J. Eggers and R. R. Henry, Evaluating runtime-compiled value-specific optimizations, Washington : University of Washington Department of Computer Science and Engineering, 1993.

[18] A. Fog, Optimizing software in C++, Lyngby: Technical University of Denmark, 2004-2014.

[19] P. Isensee, "C++ Optimization Strategies and Techniques," [Online]. Available: http://www.tantalon.com/pete/cppopt/main.htm. [Accessed 19 1 2015].

[20] S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. C. Gyllenhaal, D. M. Gallagher and W.-m. W. Hwu, "Characterizing the impact of predicated execution on branch prediction," in *MICRO 27 Proceedings of the 27th annual international symposium on Microarchitecture*, New York, 1994.

[21] J. Ruttenberg, G. R. Gao, A. Stoutchinin and W. Lichtenstein, "Software pipelining showdown: optimal vs. heuristic methods in a production compiler," in *PLDI '96 Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation* , New York, 1996.

[22] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle and P. G. Kjeldsberg, "Data and memory optimization techniques for embedded systems," *ACM Transactions on Design Automation of Electronic Systems,* vol. 6, no. 2, pp. 149-206, 2001.

[23] M. Jahre and L. Natvig, "Performance Effects of a Cache Miss Handling Architecture in a Multi-core Processor," in *NIK*, Oslo, 2007.

[24] R. H. Saavedra and R. H. Smith, "Measuring cache and TLB performance and their effect on benchmark runtimes," *Computers, IEEE Transactions on,* vol. 44, no. 10, pp. 1223-1235, 1995.

[25] J. Xu, Q. Hu, W.-C. Lee and D. L. Lee, "Performance evaluation of an optimal cache replacement policy for wireless data dissemination," *Knowledge and Data Engineering, IEEE Transactions on (Volume:16 , Issue: 1) ,* vol. 16, no. 1, pp. 125-139, 2004.

[26] R. Sheikh and M. Kharbutli, "Improving cache performance by combining cost-sensitivity and locality principles in cache replacement algorithms," in *Computer Design (ICCD), 2010 IEEE International Conference on* , Amsterdam, 2010.

[27] S. Cepeda, "What you Need to Know about Prefetching," Intel, 8 2009. [Online]. Available: https://software.intel.com/en-us/blogs/2009/08/24/what-you-need-to-know-about-prefetching.

[28] C. Tien-Fu and J.-L. Baer, "A performance study of software and hardware data prefetching schemes," in *Computer Architecture, 1994., Proceedings the 21st Annual International Symposium on* , Chicago, IL, 1994.

[29] D. F. Bacon, S. L. Graham and O. J. Sharp, "Compiler transformations for high-performance computing," *ACM Computing Surveys (CSUR),* vol. 26, no. 4, pp. 345-420, 1994.

[30] I. Park, H. Lee and H. Lee, "Software optimization for embedded communication system," in *Information Networking (ICOIN), 2013 International Conference on* , Bangkok, 2013.

[31] P. V. Joshi and K. S. Gurumurthy, "Analysing and Improving the Performance

of Software Code for Real Time Embedded Systems," in *Devices, Circuits and Systems (ICDCS), 2014 2nd International Conference on* , Combiatore, 2014.

[32] G. Stitt, R. Lysecky and F. Vahid, "Dynamic hardware/software partitioning: a first approach," in *DAC '03 Proceedings of the 40th annual Design Automation Conference* , New York, 2003.

[33] R. Ferreira, V. Duarte, W. Meireles, M. Pereira, L. Carro and S. Wong, "A just-in-time modulo scheduling for virtual coarse-grained reconfigurable architectures," in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 2013 International Conference on* , Agios Konstantinos , 2013.

[34] "Robot framework," [Online]. Available: http://robotframework.org/. [Accessed 16 2 2015].

[35] "Jenkins," [Online]. Available: http://jenkins-ci.org/. [Accessed 16 2 2015].

[36] "Perf-wiki," [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page. [Accessed 16 2 2015].